# Optimizing Incremental Scope-bounded Checking with Data-flow Analysis

Danhua Shao Divya Gopinath Sarfraz Khurshid Dewayne E. Perry
Electrical and Computer Engineering Department
The University of Texas at Austin
Austin, TX 78712, USA
{dshao, dgopinath, khurshid, perry}@ece.utexas.edu

***Abstract*— We present a novel approach to optimize incremental scope-bounded checking of programs using a relational constraint solver. Given a program and its correctness specification, scope-bounded checking encodes control-flow and data-flow of *bounded* code segments into declarative formulas and uses constraint solvers to search for correctness violations. For non-trivial programs, the formulas are often complex and represent a heavy workload that can choke the solvers. To scale scope-bounded checking, our previous work introduced an *incremental* approach that uses the program's *control-flow* as a basis of partitioning the program and generating several sub-formulas, which represent simpler problem instances for the underlying solvers. This paper introduces a new approach that uses the program's *data-flow*, specifically *variable-definitions*, as a basis for incremental checking. Experimental results show that the use of data-flow provides a significant reduction in the number of variables in the encoded formulas over the previous control-flow-based approach, thereby further improving scalability of scope-bounded checking.**

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

## I. INTRODUCTION

In software verification, *scope-bounded* checking [3, 7, 9, 14, 16, 22] of programs has become an effective technique for finding subtle bugs. Given bounds (that are iteratively relaxed) on input size [3] and length of execution paths [14], a program and its correctness specifications are translated into a formula, which is solved using off-the-shelf solvers [12, 27, 30]. A solution to the formula usually represents a counterexample to the correctness specification.

Previous work [9, 11, 20, 37] developed an approach based on the Alloy specification language (first-order logic based on sets and relations) and the Alloy Analyzer [19] for scope-bounded checking of Java programs. Given a procedure *Proc* in Java and its pre-condition *Pre* and post-condition *Post* in Alloy, the approach solves the following formula [20, 37]: $Pre \wedge translate(Proc) \wedge \neg Post$ . Given bounds on loop unrolling (and recursion depth), the *translate()* function encodes both control-flow and data-flow of the bounded code fragment into an Alloy 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 and is solved by off-the-shelf SAT solvers used by the Alloy Analyzer. A solution to this formula corresponds to an execution path in *Proc* that satisfies *Pre* but violates *Post*, i.e., a counterexample to the correctness property.

The scalability and effectiveness of scope-bounded checking in bug finding critically depends on the capabilities of the underlying constraint solvers. The traditional approaches [9, 11, 20, 37] translate the bounded code segment of the *whole* program into *one* input formula. For non-trivial programs, the translated formulas can be quite complex and the solvers can fail to find a counterexample in a desired amount of time. When a solver times out, typically there is no information about the likely correctness of the program checked or the coverage of the analysis completed.

Recently, we introduced an *incremental* approach based on the program's *control-flow* to increase the efficiency and effectiveness of scope-bounded checking [33]. The key idea is to partition the set of executions of the bounded code fragment into a number of subsets and encode each subset into a sub-formula. We *split* the program into smaller sub-programs, which are checked according to the correctness specification. 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. To illustrate, let *Proc* be split into sub-programs $Sub_1$, …, $Sub_n$. Then, checking the formula $Pre \wedge translate(Proc) \wedge \neg Post$ is equivalent to checking the sub-formulas $\{Pre \wedge translate(Sub_1) \wedge \neg Post$, ……, $Pre \wedge translate(Sub_n) \wedge \neg Post\}$.

The key insight of our incremental approach is a "sliding rule" that allows controlling the complexity of the sub-formulas based on the capabilities of the underlying solvers. Our previous work [33] introduces *splitting strategies* to embody the sliding rule. However, this work uses solely the program's *control-flow* to define the strategies, and is therefore limited to the syntactical structure of the program and fails to exploit the program semantics.

Since the complexity of the formulas comes from both the *data-flow* and the *control-flow*, we hypothesize that the use of data-flow in defining splitting strategies is likely to further reduce the workload of the constraint solvers. To evaluate the hypothesis, we introduce a splitting strategy based on *variable-definition*s. Specifically, we split the program based on different definitions of the same variable into sub-programs, which leads to a reduction in the number of variables in the resulting formulas. Experimental results show that use of variable-definitions effectively reduces variables in the formulas solved by the backend constraint solvers and significantly improves scalability.

This paper makes the following contributions:

- *Incremental scope-bounded checking using data-flow*. To optimize incremental bounded-checking, we propose a splitting strategy based on data-flow, which separates different definitions of the same variable into different sub-programs.
- *Implementation*. We implement our approach using the Forge framework [9] and KodKod model finder [35].
- *Evaluation.* We compare our data-flow-based incremental approach with the traditional approach that solves one formula (that represents the entire bounded computation segment) and with our previous incremental approach that uses only control-flow as a basis of splitting, as well as with an extreme version of the control-flow based approach, which separately checks each bounded path in the program, akin to symbolic execution. Experiments show that our data-flow-based incremental approach scales the best for complex data structures. We also test the efficacy of our approach using a real world application.

## II. Example

This section presents a small example to illustrate our variable-definition-based 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()` of class `IntList`. Assume a bound of one loop unrolling on the execution length. Figure 2(a) shows the program and its *computation graph* [20] for this bound.

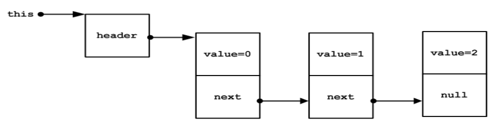
Our program splitting strategy is *variable-definition* based. Given a variable in the computation graph, we split the graph into multiple sub-graphs such that each sub-graph has at most one definition that can reach the `Exit` statement. Definitions of this variable in each of the sub-graphs are different.

In Figure 2 (a), definitions of variable `this` and `key` are empty sets {}. Definitions of variable `return` are provided by statement set {4, 8, 11}, and definitions of variable `e` are

```
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)

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

provided by statement set {1, 5, 9}. All of these definitions can reach the `Exit` statement.

Suppose we select definitions of variable `e` (which has the maximum number of definitions) to split the computation graph. We construct three sub-programs: Figure 2(b), 2(c), and 2(d). Each sub-program only contains one definition of variable `e`.

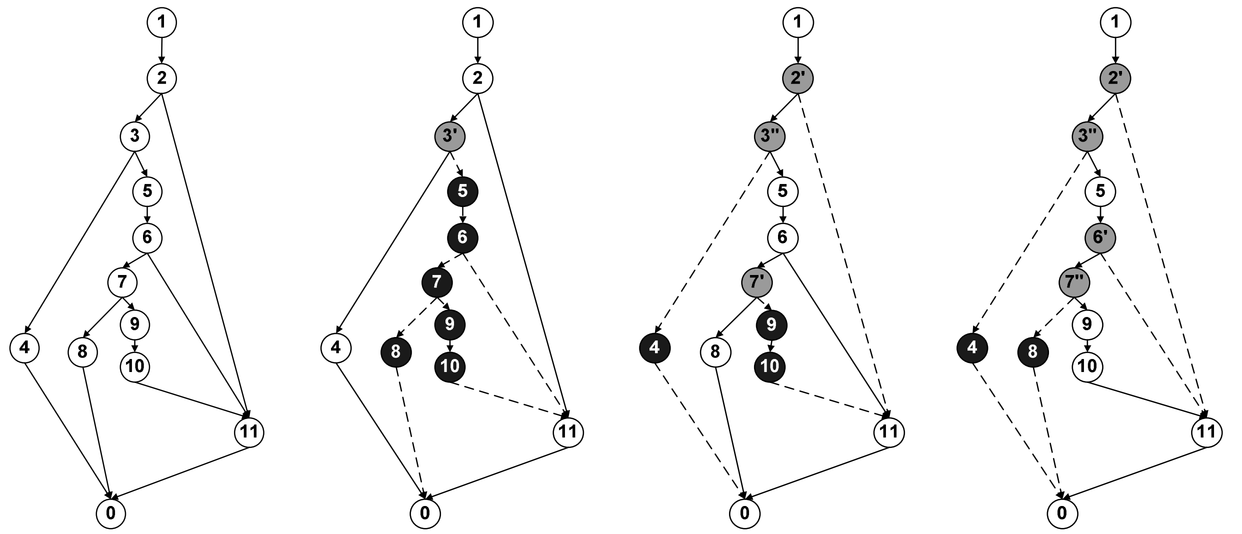
## III. Background

The goal of our computation graph splitting algorithm is to optimize traditional bounded exhaustive checking of programs using constraints in relational logic. Traditional approaches [9, 11, 20, 37] translate the whole bounded Java code segment into one relational logic formula. The conjunction of the code constraints and the negation of correctness specifications are passed to a relational logic constraint solver. Solutions are translated back to executions that violate the specification.

The translation from Java to Alloy, initially presented in the JAlloy technique [20], is based on the relational view of a program heap. This is done in three steps: (1) encoding data, (2) encoding control-flow, and (3) encoding data-flow.

Encoding data involves building a representation for classes, types, and variables in relational logic. Each class or type is represented as a set or a *domain*, which comprises of the universe of objects 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.

Data-flow is encoded as relational operations on sets and relations. Within an expression in a Java statement, field



```
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
sub1(int key)
 {
1 : Entry e = this.header;
2 : if (e != null){
3':  assume (e.value==key)
4 :   return true;

5 :
6 :
7 :
8 :

9 :

10:
    }
11: return false;
0 :}
(b)
```

```
public boolean
sub2(int key)
 {
1 : Entry e = this.header;
2': assume (e != null);
3": assume !(e.value==key);
4 :

5 :  e = e.next;
6 :  if (e != null){
7':   assume(e.value==key);
8 :   return true
     }
9 :

10:

11: return false;
0 :}
(c)
```

```
public boolean
sub3(int key)
{
1 : Entry e = this.header;
2': assume(e != null);
3": assume !(e.value==key);
4 :

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

9 : e = e.next;

10: assume(e == null);

11: return false;
0 :}
(d)
```

Figure 2. Splitting of program `contains()` based on definitions of variable `e`. *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 the programs below the computation graphs, the corresponding statements are shown in Italic. *Black nodes* denote the statements removed during splitting. Subgraph (a) is program `contains()` and its computation graph after one-round unrolling. At exit, there are three definitions of variable e: Statement 1, 5, 9. Subgraph (b) is based on definition of variable e at statement 1. Subgraph (c) is based on definition of variable e at statement 5. Subgraph (d) is based on definition of variable e at statement 9.

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 called methods.

Given a program, encoding control-flow is based on the computation graph. Each edge ($v_i \rightarrow v_j$) in the computation graph is represented as a boolean variable $E_{i,j}$, which has a value *true* when the corresponding edge is traversed. The control flow from one statement to the next sequential statement is viewed as a 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. The edge is considered traversed when there exists data that satisfies the relational formula for the edge.

The conjunction of the formulas generated by encoding the data-flow and control-flow of a statement sequence yields the formula for a path and the disjunction of the formulas for all the paths yields the formula for the code segment under analysis. The Alloy formula of the code segment, pre-condition specification, and negation of the post-condition correctness specification are conjoined and passed to an engine such as *Alloy Analyzer*. Given an input *scope* (bound on the universe of atoms/instances of each type), the engine translates the given Alloy formula into a propositional satisfiability (SAT) formula and uses off-the-shelf SAT technology to solve the formula.

*Forge* is a recently proposed framework [8, 9] which builds on the JAlloy approach and includes features which combat the shortcomings of the previous technique. It uses a custom relational engine expressly built for the application and performs optimized translations from code to logic. The Forge framework takes in code and specifications in the Forge Intermediate Language (FIR), a "relational programming language" which provides constructs in Java that support data-flow and control-flow encoding. The framework also improves translation functions for appropriate mapping of expressions from the high level Java domain to relational logic domain and vice versa. While the JAlloy tool interfaced with the *Alloy Analyzer* for translation of the constraints to boolean logic, Forge employs the *Kodkod* model finder for faster translation.

Our earlier work [33] proposed algorithms for splitting the computation graph into sub-graphs and solving them incrementally. Sub-graphs are constructed by transforming branch statements into `assume` statements.

## IV. ALGORITHM

In our previous work, we developed a *vertex*-based sub-graph analysis technique which preserves the behavioral semantics (w.r.t. to the given scope) of a program while splitting it into sub-programs. Given a vertex, we construct two sub-programs: 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 partitioning guarantees behavioral equivalence and consistency between the original program and sub-programs.

**Definition**. Given a vertex *v* in a computation graph *cg*, *go-through-sub*(*v*) is a sub-graph of *cg* that has and only has all paths that go through vertex *v*; and *bypass-sub*(*v*) is a sub-graph of *cg* that has and only has all paths that bypass vertex *v*.

The implementation and correctness proofs for the *go-through-sub*() and *bypass-sub*() functions have been discussed in our previous work [33].

In our splitting technique, vertex selection is critical. We propose a set of heuristics for vertex selection. In Figure 3, we propose a generic framework of our sub-program analysis while a splitting strategy is implemented as a splitter.

Given program *p*, we check it as following steps:

1. Translate *p* into *p'* where *p'* represents the *computation graph* [20] 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 a splitting strategy (a Splitter) to split *CG* into sub-graphs $CG_1$ $CG_2$, …, and $CG_n$

4. Recursively split each sub-graph $CG_i$ if needed;

5. With the given specifications and bounds on scope, translate each of them into a CNF formula;

```
Solution check (ComputationGraph cg, Specification spec, Bound bound, Spliter sp, int round)
{
    ArrayList<ComputationGraph> subgraph, tmp;
    ArrayList<CNF> cnf;
    Solution sol;

    subgraph.add(cg);
    for (int i = 0; i < round; i++){    //split cg round times
        for (ComputationGraph sub: subgraph) tmp.addAll(sp.split(sub));
        subgraph.clear();
        subgraph.addAll(tmp);
        tmp.clear();
    }
    for (int i = 0; i < subgraph.size(); i++){ //translate to CNF formula
        CNF f = Solver.translate(subgraph.get(i), spec, bound);
        cnf.add(f);
    }
    sort(cnf); //sort cnf formula according to clauses, variables and primary variables.
    for (int i = 0; i < cnf.size(); i++){ //solve each formula
        sol=Solver.solve(cnf.get(i));
        if (sol.statisfiable()) return sol;    //solution found
    }
    return null;
}
```

Figure 3. Sub-program-based checking algorithm

6. Sort formulas according to the number of clauses, variables, and primary variables;

7. Call solver to solve these formulas sequentially until a solution is found or all formulas are solved.

### A. *Variable-definition-based splitting strategy*

In the variable-definition-based splitting strategy, we select vertices defining values of a given variable to split a program. We separate variable definitions so that each sub-program has at most one variable definition that can reach *Exit* statement. In different sub-graphs, definitions of the variable reaching *Exit* statement are different.

**Definition**. Given a statement *s* in a program, *reach-definition*(s) = {(*var*, *Def*)}, where *Def* is a set of statements such that for each statement *d* in *Def*, variable *var* is defined at statement *d*, and there is a control-flow path from *d* to *s* such that there is no other definition of variable *var* along that path.

Given a program, its reaching definitions can be calculated by BFS (Breadth First Search) of its computation graph.

Figure 4 shows our variable-definition-based splitter. Given a program represented by a computation graph *cg*, we split it as follows:

1. Compute definitions reaching the *Exit* statement of *cg*.
2. Select the variable *v,* which has the most number of definitions reaching the *Exit* statement.
3. For each definition *d* of variable *v*, construct a sub-graph *cg*.*go-through*(*d*). This sub-graph has all the paths that visit *d*.

```
class VarDefSpliter implements Spliter {
    ......
    List<ComputationGraph> split(ComputationGraph cg)
    {
        int size = -1;
        ReachingDefinition selected;
        List<ComputationGraph> subgraph;

        for (ReachingDefinition rd : cg.getReachingDefinition(cg.ExitStmt()))//select a variable
            if (rd.getDef().size() > size){size = rd.getDef().size(); selected = rd;}
        for (Statement d: selected.getDef()) {//split based on definitions of selected variable
            sub = cg.go-through-sub(d);
            //remove other definitions that may kill current definition at exit.
            for each (ReachingDefinition rd: sub.getReachingDefinition(sub.ExitStmt())
                if (rd.getVar() == selected.getVar())
                    for each (Statement s: rd.getDef()) if (s != d) sub = sub.bypass-sub(s);
            subgraph.add(sub);
        }
        //construct the sub-graph without any definition for selected variable.
        sub = cg;
        for (Statement d: selected.getDef()) sub = sub.bypass-sub(d);
        subgraph.add(sub);
        return subgraph;
    }
}
```

Figure 4. Variable-Definition based splitting algorithm

4. For each definition *d* and its go-through sub-graph *sub*, calculate definitions of variable *v* that can reach *Exit* statement. If there is another definition *k* for the same variable below d in sub, call *sub.bypass-sub*(*k*) to remove definition *k*. Repeat this process until *d* is the only definition of variable *v* in sub-graph *sub*.
5. Call the *bypass-sub* function on every definition of variable *v* in the entire *cg,* to construct a sub-graph that has no definitions for the variable.

Steps 4 and 5 yield smaller sub-graphs from the variable-definition based splitting.

Compared with whole program analysis, the overhead of Variable-Definition based sub-program analysis composes of three parts:

- T1: Identify variable definition vertices that can reach `Exit` statement.
- T2: Construct *go-through* and *bypass* sub-graphs for these definition vertices. Given a vertex, *go-through* and *bypass* sub-graph are constructed by transforming the branches statements into `assume` statements [33].
- T3: Split the computation graph according to selected variable definitions.

Given a computation graph $CG = (V, E)$, T1 can be achieved by BFS (Breadth First Search). So the time complexity is $O(|V| + |E|)$. T2 can also be achieved by BFS, and its complexity is also $O$ $(|V| + |E|)$. T3 is the time on identifying variable definitions and branches can reach these definitions. Since variable definitions and the branch statements are subsets of $V$, T3 is not more than $(|V|^2)$. Since $|E|$ is no more than $O$ $(|V|^2)$, the summary of T1, T2, and T3 is at most $O$ $(|V|^2)$.

## B. *Branch-based splitting strategy*

In the branch-based splitting strategy introduced in our earlier work [33], we use the number of branches as the heuristic measure of the complexity of checking. To effectively divide the analysis complexity of a program, we select a vertex such that the number of branch statements in each of the sub-programs is minimized. Figure 5 shows the branch-based splitting strategy.

Given a program represented by a computation graph *cg*, we split it as follows:

1. For each vertex *v* of *cg*, construct two sub-graphs: *cg.go-through*(*v*) and *cg.bypass-sub*(*v*). Count the number of branch nodes in each sub-graph and use the larger value as the split-complexity for the splitting based on the vertex *v*.
2. Perform step 1 on all vertices and select the vertex that has the minimum split-complexity.
3. Split *cg* based on the selected vertex.

Compared with whole program analysis, the overhead of Branch-based sub-program analysis compose of three parts:

1. T1: Calculate reachability. Given a vertex, go-through and bypass sub-graph can be constructed by changing some branches statements reachable to the vertex into assume statements [33].
2. T2: Calculate number of branches for the each vertex-based splitting.
3. T3: Split according to selected vertex.

Given a computation graph $CG = (V, E)$, T1 can be achieved by BFS (Breadth First Search). So the time complexity is $O$ $(|V| + |E|)$. T2 is the sum of branches can reach each vertex. Since branch statements reaching a vertex is a subset of $V$, T2 is not more than $(|V|^2)$. With the same reason, T3 is no more than $(|V|)$. Since $|E|$ is no more than $O$ $(|V|^2)$, the summary of T1, T2, and T3 is at most $O$ $(|V|^2)$.

## V. EXPERIMENTS

We performed a set of experiments on methods of complex standard data structures to measure and compare the scalability of the sub-program-based *incremental*

```
class BranchSpliter implements Spliter {
    ......
    List<ComputationGraph> split(ComputationGraph cg)
    {
        List<ComputationGraph> subgraph;
        for (Vertex v: cg.getVertices()){
            branch1 = cg.go-through-sub(v).branches().size();
            branch2 = cg.bypass-sub(v).branches().size();
            split-complexity = max(branch1, branch2);
            if (split-complexity < current-complexity)
                {u = v; current-complexity = split-complexity;}
        }
        subgraph.add(cg.go-through-sub(u));
        subgraph.add(cg.bypass-sub(u));
        return subgraph;
    }
}
```

Figure 5. Branch-based splitting algorithm

analysis strategies with the traditional whole program analysis. We selected five methods from four standard data structure classes and compared the speedup and workload of the different splitting strategies. We also measured the performance of the strategies on a real world application, the KOA remote voting system. We piggybacked on the most recent version of the Forge tool-set [11] to implement our incremental approach. Since Forge performs modular verification of code, a method is verified as a standalone entity with respective pre and post-conditions, hence the terms "program" and "method" are used interchangeably meaning a module to be verified

## A. *Experiment with standard data structure candidates*

The candidates chosen for evaluation were – `contains` method of `Singular Linked List (LL)`, `contains` method of `Binary Search Tree (BST)`, `add` method of `Binary Search Tree`, `sort` method of `Directed Acyclic Graph (DAG)` and `insert` method of `Red Black Tree (RBT)`. `DAG.sort()` contained seeded faults and was used to compare the strategies based on the time to detect counter-examples. The other four methods were the correct versions and were used as candidates to measure the scalability of the techniques in searching the full state space. Though typical object oriented data structure methods are small in terms of the absolute number of lines of source code, on being unrolled based on the bounds, they produce large number of paths increasing the complexity of analysis. For instance, the `RBT.insert()` procedure contains 1829 lines of code after 6 unrollings.

We used the following metrics to measure the effectiveness of the splitting strategies:

- The *speed-up* obtained by the splitting strategies in comparison with the traditional analysis was calculated as the ratio of the total solving time of the back-end SAT solver in the traditional technique to the corresponding times obtained from splitting. Speedup = $T_{\text{whole-analysis}}/T_{\text{sub-program-analysis}}$.
- The size of the generated boolean formulas was measured in terms of the average number of variables across all the sub-formulas in the **C**onjunctive **N**ormal **F**orm (CNF).

The input parameters which bound the size of the verification performed by Forge are the number of *loop unrollings* (the number of times loops in the code are to be unrolled and recursive calls inlined), the *scope* (maximum number of nodes in a list, tree or graph), and *bit-width* (number of bits used to represent an integer). We used a scope of 8 for `LL.contains()` and `DAG.sort()` methods, scope of 7 for `BST.add()` and a scope of 4 for `RBT.insert()`. The number of unrolls were increased from 1 onwards up till the scope value. We checked each method using the following four splitting strategies:

- WHOLE- traditional method of checking of the entire computation graph;
- BRANCH- branch-based splitting vertex selection;
- VARDEF- variable definition based splitting vertex selection;
- PATH- splitting the computation graph such that each sub-graph comprises of a single path.

We ran the experiments on a Dual-Core 1.8GHz AMD Opteron processor with 2 GB RAM. Average values of three runs were recorded. The backend SAT solver used was MINISAT for all cases.

As seen in the results shown in Figure 6, for all the methods, the splitting strategies provided considerable speed-ups in solving times with increase in the number of unrolls. The threshold at which the splitting strategies start showing benefit over traditional analysis decreases as the complexity of the data structure increases. For instance, for singly linked list, the speed-ups obtained remain below 1 (perform worse than traditional analysis) for all strategies at 1 and 2 unrolls, whereas for red-black tree the speed-up obtained is greater than 1 even at 1 unroll.

VARDEF strategy consistently performs the best on all data structure methods. For example, in the `BST.add()` procedure, VARDEF strategy gives a 147X speedup with four unrollings while PATH and BRANCH strategies give only 60X and 2.7X speedups respectively. These results also show that VARDEF has better scalability. For example, in `BST.contains()`, when the unrolling increases from 1 to 5, speedup provided by VARDEF strategy increases 10 times, while the speedups of BRANCH and PATH strategies increase only by 3.5 and 2.8 times respectively.

For the methods of `Linked List`, `Binary Search Tree` and `Direct Acyclic Graph`, checking every path individually performs better than the BRANCH based splitting of the computation graph into sub-graphs. For instance, in `BST.add()` method , for 4 unrolls , PATH strategy gives a 60X speed-up as compared to only 2.7X speed-up of BRANCH based splitting. But for the `RBT.insert()` method the PATH strategy chokes the tool and is unable to solve all the paths even for 1 unroll ( it solves around 11406 paths in 7 hours). This is the reason why this method in Figure 6 doesn't have data for PATH strategy. The insert method comprises of 67 branches (of which 48 are conditional assignments) resulting in an exponential number of paths (approximately 14418000 paths even for 1 loop unrolling). This indicates that the performance of the PATH strategy (representative of symbolic execution) is very sensitive to increase in the complexity of the data structure which in turn translates into both syntactic and semantic complexity of the methods and the respective specifications.

For `DAG.sort()`, the speedup of VARDEF strategy is almost 540X. The reason for the high speed-up is that the fault is present in a short path. Since the sub-graphs are checked in the increasing order of complexity, the sub-graph with fault is checked early and the fault is detected very

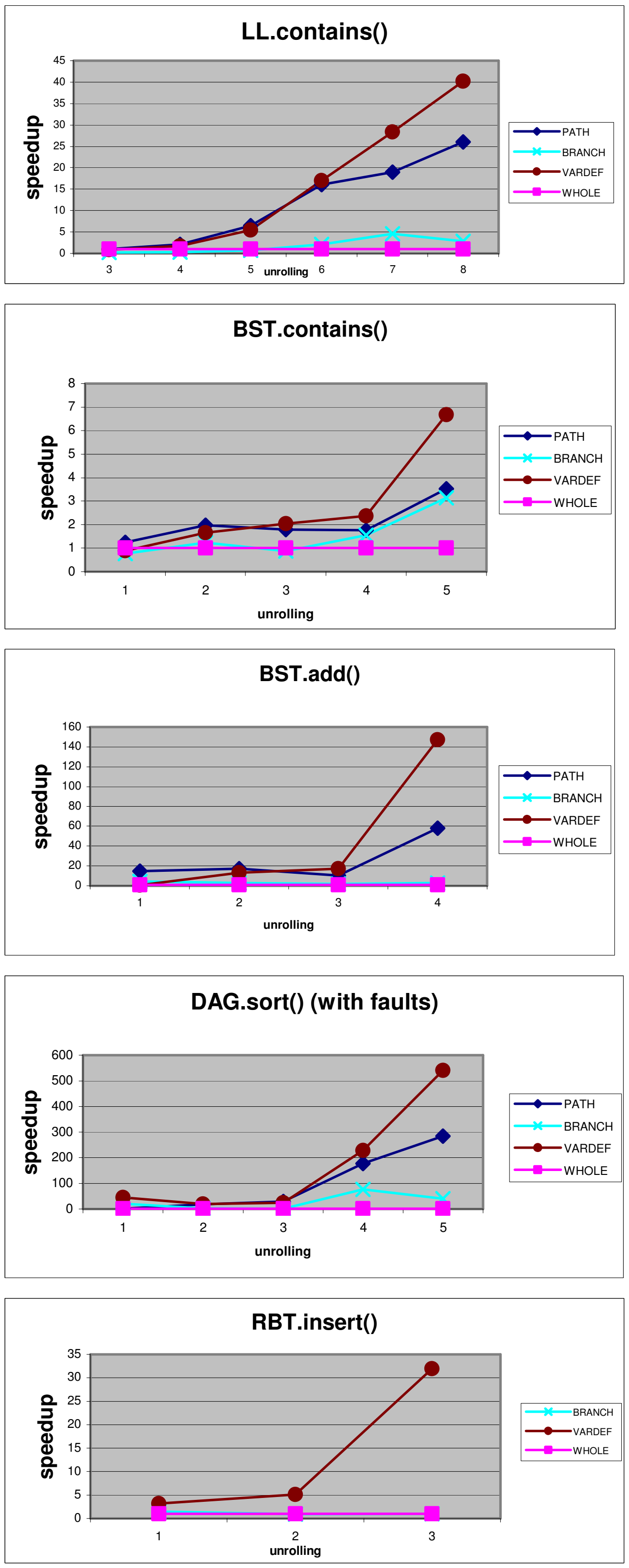


Figure 6. Speedup of sub-program analysis with Branch-based,Variable-definition-based and Path-based strategies.

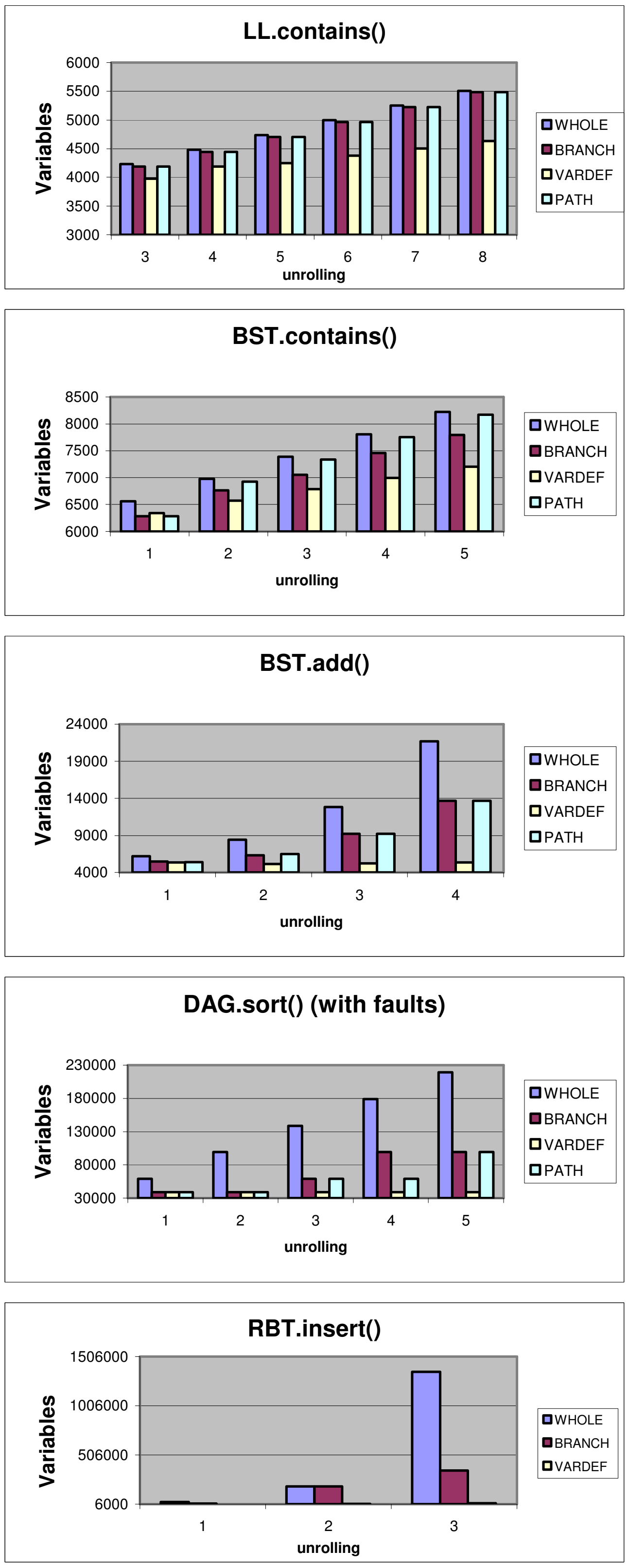


Figure 7. Average #variables of CNF formula per sub-program in Branch-based and Variable-definition-based analysis and Whole program analysis.

soon. Also, while the fault-detection time remains almost constant for the VARDEF strategy even with increase in the complexity (number of unrolls), the time taken by traditional analysis degrades. Thus, on the whole, there is an exponential increase in speed-up. The inference is that faults in short paths can be detected much faster by sub-program analysis as compared to the traditional whole program analysis. The time taken by sub-program analysis is also more resistant to increase in the size of the computation graphs and hence is more scalable than the traditional approach.

The splitting strategies incur overhead due to the static analysis involved in splitting the computation graph into sub-graphs and repeated translation of each of the sub-graphs, but yield smaller CNF formula per sub-graph which reduces the total solving time. We tabulate the constraint solving times since solving is the main bottleneck for SAT-based approaches. While there is translation overhead that contributes to the overall cost of checking, solving remains the dominant factor in the total cost. Comparison of the total checking times of the approaches shows similar scalability trends, although the absolute values of the speed-ups obtained are smaller. Note however that the maximum reduction in speedup is less than 23%, which happens in the case of `RBT.insert`(), with scope 3 and unrolling 3, where the speed-up in solving time is 43.296 while the speedup in total checking time (translation + solving) is 33.41.

To evaluate the effectiveness of the splitting strategies in reducing the size of the formulas, we also analyzed the number of variables in the CNF formula translated from specifications and sub-programs. Given a splitting strategy and a bound, we recorded the average number of variables in the CNF formulae translated from sub-programs. The results in the Figure 7 show that, with the increase in the number of loop unrollings, the number of variables in the formulas produced by the VARDEF strategy increases much slower than those of BRANCH, PATH and WHOLE program analysis. This indicates that VARDEF strategy can effectively combat state space explosion as the program scales up in size.

## B. *Experiment with a sub-system in KOA system*

In order to evaluate the efficacy of the incremental approach on a full-fledged application with sufficient magnitude, a case-study was performed on the KOA Voting application. The Dutch Tally subsystem in KOA contains JML annotated methods, earlier checked using ESC/Java static checker and JMLForge [10]. It comprises of 8 main classes. We used 67 methods from these classes for our analysis and used a scope of 2, bit width of 3 and number of unrolling 1 (which were determined to be the minimum bounds required to detect counter-examples in these methods [10]). Since there are 201 speed-up results in total, instead of tabulating all of them, we present below a summary of our analysis of the results.

The splitting strategies did not result in speed-ups over the traditional technique on *all* methods. The methods wherein the whole program analysis performed better had very less lines of code (1.4 lines of code on an average). In such methods, there wasn't much scope for dividing the computation graph into sub-graphs to optimize performance. Thus the overhead of the analysis for splitting the control flow graph overshadowed the benefit obtained by solving smaller sub-graphs. On the other hand, even a small increase in the complexity and size of the methods degraded the performance of the traditional analysis and it had the worst performance amongst the three approaches. For instance, for `KiesKring.make`(), a method with 6 lines of code, VARDEF strategy achieved 23X speedup versus the whole analysis.

We would like to highlight that this case-study was conducted more in an exploratory fashion to study the applicability of the splitting strategies in a real-world domain. This is in line with our aim to come up with a sliding rule for the strategy to be used for checking a method based on different criteria. Incremental checking provides benefits to applications with significant semantic and syntactic complexity. We selected the KOA application to serve as a benchmark for comparison with earlier evaluations done using Forge. We are also working on other applications such as the Intentional Naming System [11], comprising of complex data structures and methods.

## VI. Related work

Our work is based on previous research that models a heap-manipulating procedures using Alloy and finds counterexamples using SAT. Jackson et al. [20] 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. [37] optimized the translation to boolean formulas by using a special encoding of functional relations. Dennis et al. [9] provided explicit facilities to specify imperative code with first-order relational logic and used an optimized relational model finder [35] 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.

Our previous work on incremental scope-bounded checking [33] used *control-flow* as the basis of a splitting strategy. Specifically, we use the number of *branches* as a heuristic to compute an *analysis complexity metric* of a program. We split a program into two sub-programs so that the number of branch statements in each of sub-programs is minimized. Evaluations with Java library procedures showed the strengths and weaknesses of the branch-based splitting strategy. On the positive side, it can effectively divide the workload to backend SAT solver and achieve a high speed-up over the traditional whole program analysis. For example, with 3 loop unrolling and 7 nodes, the speedup of checking `add()` of `BinarySearchTree` is 12.16X. However, on the negative side, it does not exhibit much scalability. For example, for the `contains()` method of `BinarySearchTree`, the speedup only increases from 3.42X to 4.94X as the program size increases from 4 loop unrollings to 8 loop unrollings.

DynAlloy [13] 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 [21] or Korat [3] 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 [22, 23]. Guiding symbolic execution using concrete executions is rapidly gaining popularity as a means of scaling it up in several recent frameworks, most notably DART [15], CUTE [32], and EXE [4]. 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 [3]. While, in principle, the use of preconditions written as Java predicates allows symbolic execution to checks programs similar to the ones we have used for evaluation, a key property of such checking is that the number of calls to the constraint solver is not simply proportional to the number of bounded execution paths of interest, rather the number of calls is proportional to the product of the paths in the precondition that return true and the paths in the method under test. The path-based approach we have used in our evaluation (Section 5) can be viewed an optimized form of symbolic execution, which minimizes the number of calls to the underlying constraint solver by encoding the pre-condition as a single formula. Indeed, our incremental approaches are 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 [17, 28]. Recent advances in software model checking [14, 38] 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 [34] have been used to reduce workload of bounded verification. Dolby et al. [11] and Saturn [39] perform slicing at the logic representation level. Millett et al. [29] slice Promela programs for SPIN model checker [17]. Visser et al. [38] and Corbett et al. [4] 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 the 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, 32] and recent variants [26], 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.

## VII. Conclusions

Scalability is a key challenge for scope-bounded checking. For non-trivial programs, the formulas translated from control-flow and data-flow can be quite complex and the ensuing heavy workload can choke the solvers. Our previous work used control-flow as the basis for an incremental approach to scope-bounded checking by splitting program into smaller sub-programs and checking each sub-program separately, and demonstrated significant speed-ups over the traditional approach. This paper introduces the use of data-flow to optimize the incremental approach, specifically using a splitting strategy based on variable definitions. Experiments show that for programs with sufficient size and complexity, our use of variable definitions improves the scalability of the incremental approach; it effectively reduces the complexity of the ensuing formulas and provides more efficient analysis.

In general, incremental checking of programs opens up the following avenues for future work. In ongoing work, we are exploring strategies for applying semantic and syntactic analysis based splitting algorithms in tandem such that customized splitting techniques could be used which strike a trade-off between reducing the complexity of the resulting constraints and minimizing the translation time overhead. Since sub-graphs produced by the splitting algorithms are syntactically and semantically independent of each other, we also propose to combine incremental and parallel algorithms to scale up scope bounded checking. For applications with complex pre and post condition specifications, slicing of specifications based on the control flow graph splitting and specification driven control flow graph slicing are two significant areas for future work.

## REFERENCES

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In Proc. of *17th ACM Symposium on Operating Systems (SOSP)*, Kiawah Island, December 1999.

[2] C. Barrett, D. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In Proc. of *14th International Conference on Computer-Aided Verification (CAV)*, 2002.

[3] C. Boyapati, S. Khurshid and D. Marinov. Korat: Automated Testing Based on Java Predicates. In *Proc. of ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2002.

[4] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically Generating Inputs of Death. In Proc. of *the 13th ACM Conference on Computer and Communications Security(CCS)*, 2006

[5] Y. Cheon, G.T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Proc. of *the 16th European Conference on Object-Oriented Programming(ECOOP)*, 231-255, 2002.

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

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

[8] Gregory D. Dennis. Relational Framework for Bounded Program Verification, Ph.D. Thesis, Massachusetts Institute of Technology, September 2009.

[9] G. Dennis, F. S. H. Chang, and D. Jackson. Modular verification of code with SAT. In Proc. of *International Symposium on Software Testing and Analysis (ISSTA)*, 2006.

[10] G. Dennis, K. Yessenov, and D. Jackson. Bounded Verification of Voting Software. *Second IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, 2008

[11] J. Dolby, M. Vaziri, and F. Tip. 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

[12] N. Eén, and N. Sörensson. An extensible SAT solver. In Proc. of *the 6th International Conference on Theory and Applications of Satisfiability Testing(SAT)*, 2003

[13] M. F. Frias, J. P. Galeotti, C. G. López Pombo, and N. M. Aguirre. DynAlloy: upgrading alloy with actions. In Proc. of *International Conference on Software Engineering (ICSE)*, 2005.

[14] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In Proc.of *ACM Symposium on Principles of Programming Languages (POPL)*, 1997.

[15] P. Godefroid, N. Klarlund, and K. Sen. *DART: Directed automated random testing. In Proc. of ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2005.

[16] C. Heitmeyer, J. James Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*. Vol. 24, No. 11, 927-948, 1998.

[17] G. J. Holzmann. The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, 2004.

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

[19] D. Jackson. Software Abstractions: logic, language, and analysis. MIT Press, Cambridge, MA, 2006.

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

[21] S. Khurshid and D. Marinov. TestEra: Specification-based Testing of Java Programs Using SAT. *Automated Software Engineering Journal*, Vol.11, No. 4. October 2004.

[22] S. Khurshid, C. Pasareanu and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In Proc. of the *9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2003.

[23] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, Volume 19, Issue 7, July 1976.

[24] J. Kiniry, A. Morkan, D. Cochran, F. Fairmichael, P. Chalin, M. Oostdijk, and E. Hubbers. The KOA remote voting system: A summary of work to date. Proc.of *Trustworthy Global Computing (TGC)*, 2006

[25] N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. In Proc. of *5th International Conference on Implementation and Application of Automata*, 2000.

[26] V. Kuncak. "Modular Data Structure Verification," Ph.D. thesis, EECS Department, Massachusetts Institute of Technology, 2007.

[27] M. Leonardo and N. Bjørner. Z3: An Efficient SMT Solver. In Proc. of *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

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

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

[30] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In Proc. of *39th Design Automation Conference (DAC)*, 2001

[31] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Volume 24, Issue 3: 217 – 298, 2002

[32] K. Sen, D. Marinov, and G. Agha. 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.

[33] D. Shao, S. Khurshid, and D. E. Perry. An incremental approach to scope-bounded checking using a lightweight formal method. The *16th International Symposium on Formal Methods (FM)*, 2009.

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

[35] E. Torlak and D. Jackson. Kodkod: A Relational Model Finder. In Proc.of *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2007.

[36] E. Uzuncaova and S. Khurshid. Program Slicing for Declarative Specifications. *14th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)* Poster Paper. Portland, OR. November 2006.

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

[38] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. In Proc. of *International Conference on Automated Software Engineering (ASE)*, 2000

[39] Y. Xie, and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 29, Issue 3, 2007.