# A Case for Using Data-flow Analysis to Optimize Incremental Scope-bounded Checking (Informal Extended Abstract)

Danhua Shao Divya Gopinath Sarfraz Khurshid Dewayne E. Perry
The University of Texas at Austin
`{dshao, dgopinath, khurshid, perry}@ece.utexas.edu`

**Abstract.** 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. We have developed a new approach that optimizes incremental checking using the program's *data-flow*, specifically *variable-definitions*. We expect that splitting different definitions of the same variable into sub-programs will reduce the number of variables in the resulting formulas and the workload to the backend solvers will be effectively reduced.

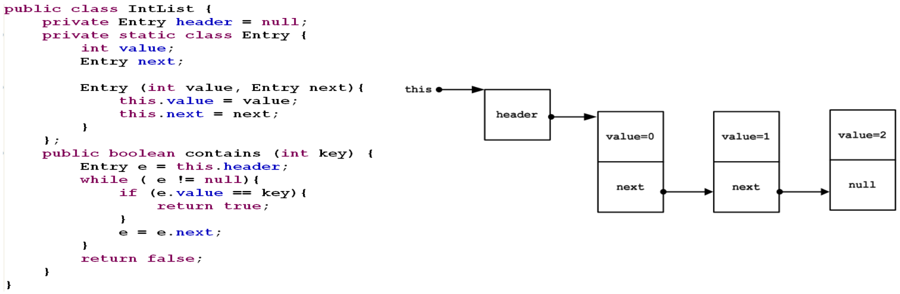
## 1 Introduction

In software verification, *scope-bounded* checking [2] of programs has become an effective technique for finding subtle bugs. Given bounds (that are iteratively relaxed) on input size and length of execution paths, the code of a program is translated into a relational logic formula, and a conjunction of this formula with the negation of the post condition specification $Pre \wedge translate(Proc) \wedge \neg Post$ is solved using off-the-shelf SAT solvers. A solution to this formula corresponds to a counterexample.

Traditional scope-bounded checking [1] translates 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 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 [3]. 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.

The splitting strategy in our previous work [3] focuses solely on the program's *control-flow,* and is therefore limited to the syntactical structure of the program and fails to exploit the program semantics.



```
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).

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. 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 sub-formulas. The rationale behind this is that decrease in the number of definitions for a variable would reduce the number of intermediate variable names and thus the number of frame conditions introduced in data flow encoding.

## 2 Example

Suppose we want to check the `contains()` method of class `IntList` (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 on execution length is one loop unrolling. Figure 2(a) shows the program and its *computation graph* [2] for this bound.

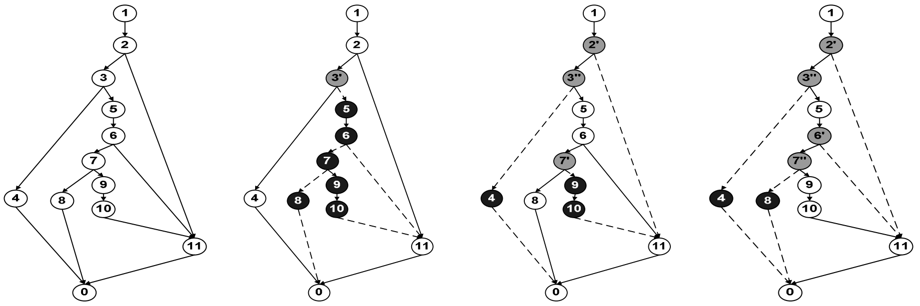
Our program splitting strategy is *variable-definition* based. Given a variable in the computation graph, we split the graph into multiple sub-graphs such that each sub-graph has at most one definition for the variable ,that can reach the exit statement.The definition of this variable in each sub-graph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

In Figure 2 (a), the definition of variable this and key is empty set {}. Definitions of variable return is statement set {4, 8, 11}, and definition of variable e is statement set {1, 5, 9}. All of these definitions can reach the exit statement.

Suppose we select definitions of variable e (the most modified variable) 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.

## 3 Summary

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



```
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 programs below computation graph, the corresponding statements are show 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 at statement 1. Subgraph (c) is based on definition at statement 5. Subgraph (d) is based on definition at statement 9.

heavy workload can choke the solvers. Our previous work used control-flow as a basis of an incremental approach to scope-bounded checking by splitting the program into smaller sub-programs and checking each sub-program separately, and demonstrated significant speed-ups over the traditional approach. We recently developed a new splitting strategy based on data-flow, specifically variable definitions, to optimize the incremental approach. We believe that use of variable definitions can effectively reduces the number of variables the complexity of the ensuing formulas and provides more efficient analysis.

**Acknowledgment:** This material is based upon work funded in part by NSF (Grants IIS-0438967, CCF-0702680, and CCF-0845628) and AFOSR (FA9550-09-1-0351).

## References

[1] G. Dennis, F. S. H. Chang, and D. Jackson. Modular verification of code with SAT. In *ISSTA* 2006.

[2] D. Jackson, and M. Vaziri. Finding bugs with a constraint solver. In *ISSTA* 2000.

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