DAG Inlining: A Decision Procedure for Reachability-Modulo-Theories in Hierarchical Programs

Akash Lal Shaz Qadeer
Microsoft Research
{akashl, qadeer}@microsoft.com

Abstract

A hierarchical program is one with multiple procedures but no loops or recursion. This paper studies the problem of deciding reachability queries in hierarchical programs where individual statements can be encoded in a decidable logic (say in SMT). This problem is fundamental to verification and most directly applicable to doing bounded reachability in programs, i.e., reachability under a bound on the number of loop iterations and recursive calls.

The usual method of deciding reachability in hierarchical programs is to first inline all procedures and then do reachability on the resulting single-procedure program. Such inlining unfolds the call graph of the program to a tree and may lead to an exponential increase in the size of the program. We design and evaluate a method called DAG inlining that unfolds the call graph to a *directed acyclic graph* (DAG) instead of a tree by sharing the bodies of procedures at certain points during inlining. DAG inlining can produce much more compact representations than tree inlining. Empirically, we show that it leads to significant improvements in the running time of a state-of-the-art verifier.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords Verification Condition generation, Bounded Model Checking, Procedure inlining, Satisfiability modulo theories, Reachability modulo theories

1. Introduction

Advances in the area of satisfiability (SAT) and satisfiability modulo theories (SMT) solvers has had a significant impact on automated software verification. SAT and SMT solvers, even though they address NP-complete problems, have earned a reputation of being robust, efficient and scalable. Their success and continuous improvements over the last few years have prompted verification tools to employ them much more directly than before.

One concrete evidence is the gaining popularity of verifiers generally referred to as *bounded model checkers* (BMC). BMC tools focus on a restricted version of full program verification: Given a user-specified bound, search all program behaviors under that bound for assertion violations. Bounding usually includes enforcing maximum number of loop iterations and recursive calls. When the operational semantics of a language can be encoded in SAT or SMT, such a bounded version of verification is a decidable problem [11]. BMC tools all operate by encoding the set of all bounded executions as a SAT/SMT formula, a process called *verification condition (VC) generation*; the difference between different tools is in how this encoding is performed.

PLDI '15, June 13–17, 2015, Portland, OR, USA.

http://dx.doi.org/10.1145/2737924.2737987

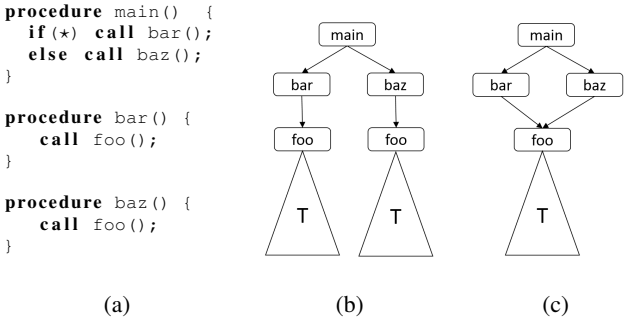

Figure 1. DAG Inlining example

Both bounded and unbounded verification problems can be reduced to reachability on *hierarchical* programs. A hierarchical program is one with possibly multiple procedures but no loops and no recursion. For bounded verification, once loops have been unrolled and recursion unfolded up to a bound, the resulting program is hierarchical. For unbounded verification, given annotations only for loops and recursive procedures, the back edges in the flow graph and the call graph can be eliminated to create a hierarchical program.

VC generation algorithms for single-procedure hierarchical programs are well understood. For hierarchical programs with multiple procedures, the unquestioned approach is to inline all procedures to result in a single-procedure hierarchical program. This process of inlining procedures, which can be thought of as unrolling the call graph of the program to a tree, has an exponential cost. The size of the resulting program can be exponential in the size of the original hierarchical program, and this directly impacts the scalability of tools.

The main contribution of this paper is a novel algorithm for VC generation of multi-procedure hierarchical programs. Instead of unrolling the call graph to a tree, we unroll it to a *directed acyclic graph* (DAG). We give conditions under which using a DAG is precise, i.e., sound and complete compared to tree inlining. Using a DAG can lead to up to exponential reductions in the size of the generated VC.

We illustrate the basic idea behind our approach using Fig. 1. We show a simple hierarchical program in Fig. 1, where the code for

`foo` has been elided away. Usual inlining, which we will refer to as tree-based inlining in this paper, proceeds by inlining `bar` and `baz` into `main`, creating two call sites of `foo`. Next, each of these call sites gets replaced with the body of `foo` and then its callees and so on. This inlining is illustrated by the tree shown in Fig. 1(b). The tree captures an unfolding of the call graph of the program and shows the amount of inlining that needs to be performed to get to a single-procedure program. `T` refers to the unfolding of the call graph reachable from `foo`. Note that two copies of `foo` and `T` are created by tree inlining.

We show that with our DAG inlining technique it is possible to unfold the call graph to a DAG. Notice that on any program execution `foo` cannot be called twice: either the execution takes the `then` branch in `main` and goes through `bar`, or it takes the `else` branch and goes through `baz`. In such a case, DAG inlining will *share* the body of `foo` to obtain the DAG shown in Fig. 1(c). In general, we show how to merge bodies of procedure instances that are known to be never taken on the same execution. This merging can have large benefits in terms of compressing the VC needed to represent all behaviors of a hierarchical program. In our example, the size of `T` can be arbitrarily large, making the savings also arbitrarily large.

We illustrate the potential for speedups using the contrived example of Fig. 2, which is an extension of the program in Fig. 1. The program is parameterized by the value of "N". Program execution starts in `main` and proceeds through a chain of procedures `P0`, `P1`, $\cdots$, `PN`, and the only assertion is in `PN`. It is easy to see that this program is correct; the invariant at the beginning of procedure `Pi` is that `g` $==$ `i`.

Fig. 3 shows the running time of two tools: CBMC [6, 7] and Corral [14] that each employ tree inlining (in their own ways) as the value of N is increased. Note that the Y-axis is on log scale. CBMC performs better than Corral but it is clear that both have exponential scalability with respect to the value of N. Our method, shown as "DI", which uses DAG inlining, has linear scalability. The reason for improved scalability is simply that the size of the VC generated by DI scales linearly with N for this program. The structure of inlining is also shown in Fig. 2. Both CBMC and Corral construct the tree inlined version, which is exponentially sized in N.

DAG inlining is based on the idea of merging procedure instances that can never be called on the same execution. A natural question to ask is if there is enough opportunity to merge in real programs? Our evaluation shows that despite of pruning optimizations performed by state-of-the-art verifiers, DAG inlining produces a VC that is three times as compact. Without pruning (which is true of verifiers that eagerly inline all procedures), DAG inlining can lead up to 200 times compression, as shown in Fig. 4.

Producing a compact VC is only one step of the solution. Eventually, we must evalute the time taken by the SAT/SMT solver to discharge the VC. Our evaluation shows that DAG inlining is able to solve many more verification problems than tree inlining within the timeout budget. On instances that both solved within the timeout, DAG inlining was approximately twice as fast.

This paper makes the following contributions.

- We give a VC generation algorithm to decide reachability in hierarchical programs. The algorithm is based on the novel idea of DAG inlining.
- We give an extensive evaluation of DAG inlining compared to tree inlining using a state-of-the-art verifier on real data. Our experiments show that DAG inlining can be much faster than tree inlining, often allowing verification to finish when tree-inlining would time out.

```
var g: int;

procedure main()  {
  g := 0;
  if(*) call P0();
  else call P0();
}

procedure Pi() {
   g := g + 1;
   if(*) call Pi+1();
   else call Pi+1();
}

procedure PN() {
   assert g == N;
}
```

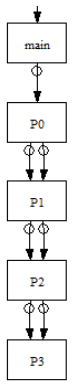

Figure 2. A program parameterized by the value of N and an illustration of DAG inlinig for N=3

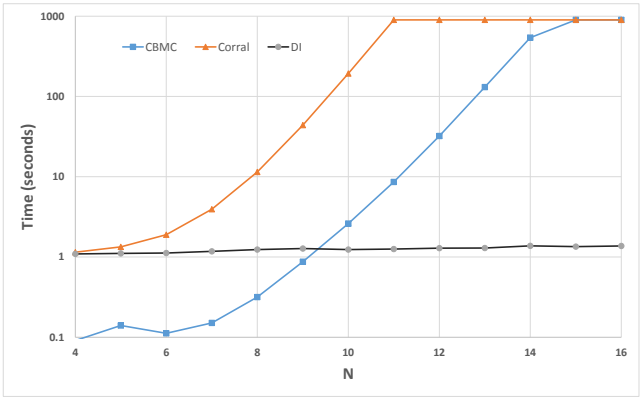

Figure 3. Comparison of BMC tools against DAG Inlining (DI) on the program of Fig. 2, under a timeout of 900 seconds.

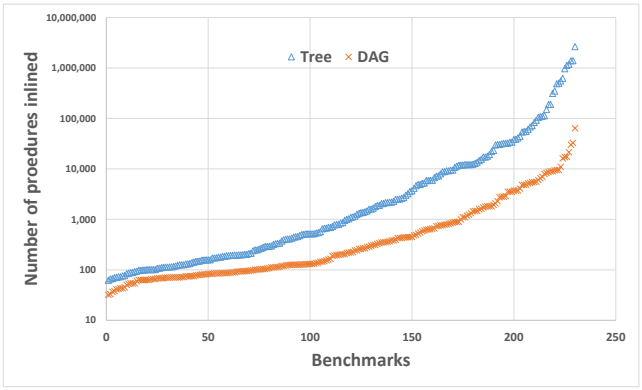

Figure 4. Tree vs. DAG sizes

2. Overview

A *dynamic* instance of a procedure is a procedure qualified by its call stack. A procedure can have as many dynamic instances as the number of its calling contexts.

As mentioned in the previous section, merging is only done for dynamic instances of the same procedure when they are *disjoint*,

i.e., they can never be taken on the same program execution. In this section, we first show commonly occurring programming patterns that create opportunities for merging in hierarchical programs, even when they are obtained from real programs after unrolling loops and unfolding recursion. These patterns were observed in our experiments on real world programs.

Consider a hierarchical program with the following statement:

```
switch(...) {
   case A: foo1(); break;
   case B: foo2(); break;
   case C: foo3(); break;
   ...
}
```

Because there are no loops and recursion, we know that once a case block is entered, there is no way to enter a different case block. This creates opportunities for disjointness, either immediately if some $\texttt{foo}_i$ is the same procedure as $\texttt{foo}_j$ for $i \neq j$, or transitively, if some $\texttt{foo}_i$ and $\texttt{foo}_j$ for $i \neq j$ end up calling the same procedure `bar`; those dynamic instances of `bar` will be disjoint.

Switch statements can, of course, occur for a variety of reasons, e.g., (a) deciding on the type of a request and dispatching to the appropriate handler, or (b) a test harness that dispatches to multiple entrypoints, or (c) IR simplification such as resolving a function pointer to its possible targets, and so on. Note that there is nothing special about using a switch statement. Even branches of an `if` condition create opportunities for disjointness in a similar manner.

Such disjointness is not a fictitious creation because we consider "hierarchical" programs. While in real (loopy) programs, it is natural to have branches inside a loop and executions that take both sides of the branch, we can still make a finer distinction. The *same* iteration of the loop cannot take both sides of a branch and this still creates disjointness that is specific to each iteration of the loop. This observation naturally falls out once loops are unrolled, which is done anyway for doing bounded reachability.

Deciding disjointness Our analysis does not rely on any particular programming idiom or pattern to decide disjointness. Instead, we analyze the control structure of the program as an (acyclic) push-down system to decide if two dynamic procedure instances are disjoint. Our method can be implemented efficiently. It requires a pre-processing time that is quadratic in the size of a single procedure and linear in the number of procedures of a program. Then disjointness of two dynamic instances can be answered in time proportional only to the length of their calling context.

Merging strategies It can often be the case that there are several possibilities of merging dynamic instances of a procedure. An approach that picks disjoint instances randomly and merges them may not be optimal. For instance, consider a procedure with the control-flow graph shown in Fig. 5(a) and suppose that nodes A–D each call the same procedure. We examine which of these calls can be merged when inlining is performed.

Fig. 5(b) is the induced *conflict* graph on the call nodes: there is an undirected edge between n_1 and n_2 if they are not disjoint. Two calls can be merged (i.e., share the body of the callee) if there is no edge between them in the conflict graph. If we decide to merge B and C then we end up with the graph Fig. 5(c) where there is no further opportunity to merge. On the other hand, if we merge A and B, we can still merge C and D and end up with Fig. 5(d) that has more compression.

In general, one strategy is to construct the conflict graph of all dynamic instances of a procedure and color it so that adjacent nodes do not have the same color. Using minimum number of colors (commonly called the *graph coloring problem*) leads to maximum compression: nodes of the same color should be merged. This strategy, while it performs the best when the program needs to be fully inlined, may not be the best when only parts of the program are inlined. We give multiple strategies to pick instances that should be merged in Section 3.4 and evaluate them in Section 4.

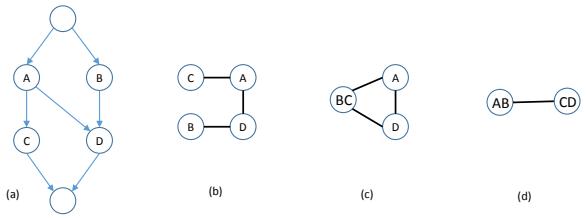

Figure 5. (a) The control-flow graph of a procedure will four calls to the same callee; (b) the induced conflict graph of the callee; (c) conflict graph after merging B with C; (d) conflict graph after merging A with B and C with D.

```
var g: int;

procedure main(v1: int, v2: int)  returns (r: int)
{ var c: bool;
  if (c) {L1: call r := foo(v1);}
  else {L2: call r := foo(v2);}
}

procedure foo(a: int) returns (b: int)
{   b := a + 1;   }
```

Figure 6. A program to illustrate VC generation

VC generation with merging Consider the program shown in Fig. 6. `main` has two disjoint calls to `foo`, labelled with `L1` and `L2`, respectively. The goal of VC generation is to create a constraint between the input state of the program and its output state.

We first create a VC for each procedure individually (called a procedural VC, or pVC). The pVC of `main` is $\phi_{\texttt{main}}(v_1, v_2, r) = (c \wedge E_0) \vee (\neg c \wedge E_1)$, where c is the value on which the branch is taken and E_i is a place holder indicating if the call at label $\texttt{L}_{i+1}$ is taken. Note that the relationship between the return value r and input parameters v_1 and v_2 is unconstrained in $\phi_{\texttt{main}}$. The pVC of foo is $\phi_{\texttt{foo}}(a, b) = (b == a + 1)$.

We further introduce constants N_i, one for each procedure instance that is inlined, to indicate when execution enters that instance. Tree inlining produces the following VC:

$$
\begin{array}{lll}
 & N_0 & \text{// execution starts in main} \\
\wedge & N_0 \Rightarrow ((c \wedge E_0) \vee (\neg c \wedge E_1)) & \text{// pVC of main} \\
\wedge & E_0 \Rightarrow (N_1 \wedge v_1 == a_1 \wedge r == b_1) & \text{// callsite L1} \\
\wedge & E_1 \Rightarrow (N_2 \wedge v_2 == a_2 \wedge r == b_2) & \text{// callsite L2} \\
\wedge & N_1 \Rightarrow (b_1 == a_1 + 1) & \text{// pVC of foo} \\
\wedge & N_2 \Rightarrow (b_2 == a_2 + 1) & \text{// pVC of foo}
\end{array}
$$

In this VC, the N_i variables guard their pVCs. The call-site constants E_i guard a constraint for the call-site. For instance, setting E_0 to *true* indicates that the call at `L1` is taken, which is enforced by asserting N_1 and an equality between formals of `foo` and actuals of the call.

$$
\begin{array}{rcll}
P & \in & Prog & ::= \quad (gs, ls, ps, init, bs, ts) \\
v, g, h & \in & Global & \\
gs & \in & 2^{Global} & \\
v, l, k & \in & Local & \\
ls & \in & 2^{Local} & \\
p, q & \in & Proc & \\
ps & \in & Label \rightarrow Proc & \\
init & \in & Proc \rightarrow Label & \\
bs & \in & Label \rightarrow Stmt & \\
ts & \in & Label \rightarrow 2^{Label} & \\
st & \in & Stmt & ::= \quad assume\ e \mid v := e \mid call\ p \\
i, f, x, y, b & \in & Label & \\
xs & \in & 2^{Label} & \\
e & \in & Expr &
\end{array}
$$

Figure 7. Program syntax

DAG inlining, when it decides to merge the two calls to `foo`, will produce the following VC:

$$
\begin{array}{lll}
 & N_0 & \text{// execution starts in main} \\
\wedge & N_0 \Rightarrow ((c \wedge E_0) \vee (\neg c \wedge E_1)) & \text{// pVC of main} \\
\wedge & E_0 \Rightarrow (N_1 \wedge v_1 == a_1 \wedge r == b_1) & \text{// callsite L1} \\
\wedge & E_1 \Rightarrow (N_1 \wedge v_2 == a_1 \wedge r == b_1) & \text{// callsite L2} \\
\wedge & N_1 \Rightarrow (b_1 == a_1 + 1) & \text{// pVC of foo}
\end{array}
$$

The difference with tree inlining is that both E_0 and E_1 enable N_1 and share the pVC of `foo`. Note that this is semantically equivalent to the tree-inlined version: setting c to *true* in each case simplifies the formula to $r == v_1 + 1$, and setting c to *false* in each case simplifies the formula to $r == v_2 + 1$. The key reason why the merging works is that both E_0 and E_1 (and also both N_1 and N_2) need never be simultaneously *true*.

3. DAG Inlining

We describe our VC generation algorithm for hierarchical programs based on DAG Inlining using the simple programming language shown in Fig. 7. A program P is a tuple $(gs, ls, ps, init, bs, ts)$. The finite set gs contains the names of global variables. The finite set ls contains the names of local variables. The control flow of the program is defined over the finite set of labels in $Label$. These labels are partitioned among the procedures in the set $Proc$. This partition is specified via the map ps from $Label$ to $Proc$. The map $init$ provides for each procedure $p \in Proc$ the initial label from which a call to p begins execution. For each label $x \in Label$, the map bs provides the code statement executed at x and the map ts the set of labels to one of which control moves nondeterministically after executing the statement $bs(x)$; if the set $ts(x)$ is empty, then control returns to the caller of $ps(x)$. We assume that the set of labels of a procedure is closed under control transfers, that is, for all $p \in Proc$, $x \in Label$ and $y \in ts(x)$, we have $ps(init(p)) = p$ and $ps(x) = ps(y)$.

Although local variables of procedures are declared at the top level in a program, dynamic instances of procedures get their own copy of storage associated with those local variables. The initial value of the global variables in the beginning of program execution and the initial value of the local variables in the beginning of a procedure execution is unconstrained (nondeterministic). To simplify presentation, we have chosen to omit input and output parameters from procedures but these can be simulated through an appropriate use of local and global variables.

A statement at a label is one of three kinds, $assume\ e$, $v := e$, and $call\ p$, where v is a variable name in $Global \cup Local$, e is an expression over $Global \cup Local$, and p is a procedure name. The semantics of these statements is standard. The statement $assume\ e$ blocks if e evaluates to false and otherwise terminates without any side effect. The statement $v := e$ assigns the result of evaluating e to v. The statement $call\ p$ calls the procedure p. While our language does not include conditional control flow, it can be encoded using nondeterminism and assume statements. Other modeling languages such as Boogie [5] also provide a *havoc* statement for updating a variable with a nondeterministic value. Since local variables are nondeterministically initialized at the beginning of a procedure call, our language can simulate *havoc* using procedure calls and the assignment statement.

We elide the syntax of expressions and the types of variables from our description; the formal requirements are mentioned elsewhere [11]. Our implementation handles all types and expressions supported by existing satisfiability-modulo-theory solvers and formalized in the SMTLIB standard, including bitvectors, integers, arrays, and datatypes.

DEFINITION 1. (REACHABILITY DECISION PROBLEM) *Given a program P and a procedure p of the program, find a terminating execution of p, i.e., an execution that returns from p.*

The problem of finding assertion violations (or violations of general safety properties) can be compiled to the reachability decision problem [13].

Let P be the program $(gs, ls, ps, init, bs, ts)$. The program P induces a *flow graph* whose nodes are the elements of $Label$ and (x, y) is an edge if and only if $y \in ts(x)$. The program P also induces a *call graph* whose nodes are the elements of $Proc$ and (p, q) is an edge if and only if there exists $x \in dom(bs)$ such that $ps(x) = p$ and $bs(x) = call\ q$. The program P is *hierarchical* if its induced flow and call graphs are acyclic.

3.1 The DAG Inlining Algorithm

The DAG inlining algorithm solves the reachability decision problem for a hierarchical program. It takes a hierarchical program P and a procedure p as input and generates a formula ϕ such that ϕ is satisfiable if and only if p has a terminating execution.

Fig. 8 shows pseudo-code for the DAG inlining algorithm. The basic data structure used by this algorithm is a DAG; the types $Node$ and $Edge$ represent the nodes and edges, respectively, of the DAG. $Root$ is the root node of the DAG. The maps Src and $Dest$ map edges to nodes; they indicate the source and destination node of an edge, respectively. The nodes represent dynamic procedure instances and the edges represent calls from the source procedure to the destination procedure. $Entry$ maps a node to the entry label of its procedure. $Callee$ maps an edge to the name of its destination procedure. $CallSite$ maps an edge to the label of the block where the corresponding call was made.

The type $Const$ refers to symbolic constants. The map $Control$ maps each edge and node to a distinct Boolean symbolic constant; this variable is used as a flag to indicate if program execution hits that particular node or edge. The maps In and Out provide the input and output interfaces of nodes and edges. The input and output interfaces of a node are sequences of symbolic constants representing the value of global variables at the input and output, respectively, of the node procedure. Intuitively, these correspond to formal input and output parameters of the procedure. Similarly, the input and output interfaces of an edge are sequences of symbolic constants representing the value of global variables before and after the call represented by the edge. Intuitively, these correspond to actual input and output arguments of the call.

The method Gen_pVC is very similar to the standard intra-procedural VC generation algorithm [4]. It generates and pushes the procedural VC (pVC) of a given procedure. We clarify the notation used in Gen_pVC. The method $Push(e)$ asserts the expression e to the solver (i.e., it conjoins e with expressions previously pushed). For a set X, let m, m_1, m_2 be maps of type $[X]Const$. The formula $m_1 = m_2$ is shorthand for $\bigwedge_{x \in X}(m_1[x] = m_2[x])$. For an expression e over variables X, $e[m]$ refers to substituting each

```
1  Input: P, p
2
3  var Root : Node
4  var Src : [Edge]Node
5  var Dest : [Edge]Node
6  var Entry : [Node]Label
7  var Callee : [Edge]Proc
8  var CallSite : [Edge]Label
9  var Control : [Node ∪ Edge]Const
10 var In : [Node ∪ Edge][Global]Const
11 var Out : [Node ∪ Edge][Global]Const
12
13 Gen_VC() {
14   var n : Node ∪ {None}
15   var d : DAG
16
17   Root = Gen_pVC(p)
18   while (dom(Src) ⊃ dom(Dest)) {
19     pick c ∈ dom(Src) \ dom(Dest)
20     pick compatible n ∈ Node ∪ {None}
21     if (n = None) {
22       n = Gen_pVC(Callee[c])
23     }
24     Dest[c] = n
25     Push(Control[c] ⇒
26             Control[n] ∧ In[c] = In[n] ∧ Out[c] = Out[n])
27   }
28   Push(Control[Root])
29 }
30
31 Gen_pVC(q : Proc) : Node {
32   var n : Node
33   var c : Edge
34   var BS : [Label]Const
35   var VS : [Label][Global ∪ Local]Const
36   var VS' : [Label][Global ∪ Local]Const
37
38   n = new Node
39   foreach (y : ps(y) = q) {
40     BS[y] = new Const
41     foreach (v ∈ gs ∪ ls) {
42       VS[y][v] = new Const
43       VS'[y][v] = new Const
44     }
45   }
46   Entry[n] = init(q)
47   Control[n] = BS[init(q)]
48   In[n] = VS[init(q)]|gs
49   foreach (v ∈ gs)
50   { Out[n][v] = new Const }
51   foreach (y : ps(y) = q) {
52     match bs(y) with
53     | assume e →
54       Push(BS[y] ⇒ e[VS[y]] ∧ VS'[y] = VS[y])
55     | v := e →
56       Push(BS[y] ⇒
57               VS'[y][v] = e[VS[y]] ∧
58               VS'[y]|gs+ls−v = VS[y]|gs+ls−v)
59     | call p →
60       c = new Edge
61       Src[c] = n
62       Callee[c] = p
63       Control[c] = BS[y]
64       In[c] = VS[y]|gs
65       Out[c] = VS'[y]|gs
66       CallSite[c] = y
67       Push(BS[y] ⇒ VS'[y]|ls = VS[y]|ls)
68
69     if (ts(y) = ∅)
70       Push(BS[y] ⇒ VS'[y]|gs = Out[n])
71     else
72       Push(BS[y] ⇒ ⋁x∈ts(y) BS[x] ∧ VS'[y] = VS[x])
73   }
74   return n
75 }
```

Figure 8. DAG inlining

```
// State: Node = {}, Edge = {}
17.  [ Root = Gen_pVC(p) ]
     // Gen_pVC invokes: [ Push(N0 ⇒ ((c ∧ E0) ∨ (¬c ∧ E1))) ]
// State: Root = n0, Node = {n0}, Edge = {e0, e1},
          Src = {e0 ↦ n0, e1 ↦ n0}, Dest = {},
          Callee = {e0 ↦ foo, e1 ↦ foo},
          Control = {n0 ↦ N0, e0 ↦ E0, e1 ↦ E1},
          CallSite = {e0 ↦ L1, e1 ↦ L2}
19.  [ pick c = e0 ]
20.  [ pick n = None ]
22.  [ n1 = Gen_pVC(foo) ]
     // Gen_pVC invokes: [ Push(N1 ⇒ (b1 == a1 + 1)) ]
24.  [ Dest[e0] = n1 ]
// State: Root = n0, Node = {n0, n1}, Edge = {e0, e1},
          Src = {e0 ↦ n0, e1 ↦ n0}, Dest = {e0 ↦ n1},
          Callee = {e0 ↦ foo, e1 ↦ foo},
          Control = {n0 ↦ N0, n1 ↦ N1, e0 ↦ E0, e1 ↦ E1},
          CallSite = {e0 ↦ L1, e1 ↦ L2}
25.  [ Push(E0 ⇒ (N1 ∧ v1 == a1 ∧ r == b1)) ]
19.  [ pick c = e1 ]
20.  [ pick n = n1 ]  // compatible choice
24.  [ Dest[e1] = n1 ]  // creates merging in the DAG structure
25.  [ Push(E1 ⇒ (N1 ∧ v2 == a1 ∧ r == b1)) ]
// State: Root = n0, Node = {n0, n1}, Edge = {e0, e1},
          Src = {e0 ↦ n0, e1 ↦ n0}, Dest = {e0 ↦ n1, e1 ↦ n1},
28.  [ Push(N0) ]
```

Figure 9. An execution of DAG Inlining over the program of Fig. 6. Instructions of the algorithm are boxed and shown with their corresponding line numbers. The value of some of the global variables is also shown at certain points during the execution.

x with $m[x]$ in e. For a set $Y \subseteq X$, $m|_Y$ refers to restricting the domain of m to Y.

We focus our attention on the top-level method Gen_VC. This method creates the DAG-inlined VC for the input procedure p through a sequence of calls to $Push$. It begins by initializing the DAG with a node and edges corresponding to the body of p and the calls from it (line 17). Then, it enters a loop which executes as long as there is an edge that has not been bound to a destination node (line 18). Such an edge, called an *open* edge, corresponds to a procedure call that has not yet been inlined. Clearly, for hierarchical programs this loop of picking open edges will terminate.

After picking an open edge c (line 19), the method Gen_VC then non-deterministically picks either the special value $None$ or a node of the DAG (line 20). If the value picked is $None$, Gen_VC invokes Gen_pVC to create a freshly-inlined copy of the target of the call (line 22). Note that the method Gen_pVC creates and returns a new node (lines 38 and 74) that is then bound to the destination of c (line 24). Further, Gen_pVC also creates additional (open) edges corresponding to the calls made by inlined procedure (lines 60–66). If, instead of $None$, an existing node n is picked at line 20, then we avoid the call to Gen_pVC and simply bind c to n. In either case, after binding the target of the call (line 24), we assert equality between formals and actuals of the call (line 25). Fig. 9 shows the execution of Gen_VC on the program of Fig. 6 that produces the DAG-Inlined VC discussed in Section 2.

Our algorithm uses non-deterministic choices for picking the open call c (line 19) and the target node n (line 20). The choice for c is completely unconstrained as the correctness of Gen_VC does not depend on it. The freedom in picking open call sites is essential for integration with state-of-the-art tools that use their

```
type DAG = (Node, 2^Node, 2^(Node × Node),
            Node → Label, Node × Node → Label)

CreateDAG() : DAG {
  let
    es = {(n, n') | ∃c ∈ dom(Dest). Src[c] = n ∧ Dest[c] = n'},
    nls = {(n, x) | Entry[n] = x},
    els = {(n, n', x^e) | ∃c.Src[c] = n ∧ Dest[c] = n' ∧ CallSite[c] = x}

  in
    return (Root, cod(Src), es, nls, els)
}

/* pick compatible n ∈ Node ∪ {None} */
pick n ∈ Node ∪ {None}
assume (n = None) ∨
       IsConsistent(CreateDAG() + (Src[c], CallSite[c], n))
```

Figure 10. Resolving line 20 of Fig. 8

own heuristics to decide the order in which to inline procedures [1, 14, 18]. The choice for n can always be $None$; if the algorithm picks $None$ each time we obtain the special case of tree inlining as every open call results in a call to Gen_pVC. Choices other than $None$ are constrained to be *compatible* with the current DAG; this notion, along with a proof of correctness that relies on it, is given in the next section.

3.2 Consistent DAG

For a set S, let $S^{\otimes}$ refer to strings of arbitrary length over the alphabet S, i.e., $S^{\otimes} = \{s_1 s_2 \cdots s_n | n \geq 0, s_i \in S\}$. Let $S^{\oplus}$ be the set of non-empty strings over S. For sets S_1 and S_2, let $S_1 \cdot S_2$ be pairwise string concatenation, i.e., $S_1 \cdot S_2 = \{s_1 s_2 \mid s_1 \in S_1, s_2 \in S_2\}$. For a binary relation R, let R^* be its reflexive transitive closure and R^+ be its non-reflexive transitive closure.

Let us fix a hierarchical program P with set of labels $Label$. Let $\Gamma = Label \cup \{b^e \mid b \in Label\}$. Intuitively, b denotes the control location just before the statement at b and b^e denotes the control location just after the statement at b. Let $\Gamma_{ret} = \{b^e \mid b \text{ has a procedure call}\}$ be a subset of Γ representing the set of return sites. A *configuration* of program P is an element of $\Gamma \cdot \Gamma_{ret}^{\otimes}$, representing a possible call stack that may arise during program execution. The transition relation $\rightsquigarrow$ of a program P is a binary relation over configurations of P. It is the smallest relation such that:

1. If block b has a non-call statement, then for any $u \in \Gamma_{ret}^{\otimes}$, $bu \rightsquigarrow b^e u$.
2. If block b has a call statement to procedure p, then for any $u \in \Gamma_{ret}^{\otimes}$, $bu \rightsquigarrow init(p) b^e u$.
3. If b_1 has a successor block b_2, then for any $u \in \Gamma_{ret}^{\otimes}$, $b_1^e u \rightsquigarrow b_2 u$.
4. If b does not have a successor block, then for any $u \in \Gamma_{ret}^{\oplus}$, $b^e u \rightsquigarrow u$.

Two configurations c_1 and c_2 of a program P are called *disjoint*, denoted by $Disj(c_1, c_2)$, if neither $c_1 \rightsquigarrow^* c_2$ nor $c_2 \rightsquigarrow^* c_1$.

Fig. 10 shows how to resolve line 20 of Fig. 8. The method $CreateDAG$ converts the imperative data structure of Fig. 8 to a mathematical representation of a DAG. A DAG is defined as a tuple $(r, N, E, \mathcal{L}_N, \mathcal{L}_E)$, where N is a set of nodes, $r \in N$ is the root node, $E \subseteq N \times N$ is a set of edges, $\mathcal{L}_N$ is a map from N to Γ and $\mathcal{L}_E$ is a map from E to Γ_{ret}. Further, N and E together must define an acyclic graph. As with the imperative notation, each node $n \in N$ corresponds to (possibly multiple) dynamic instances of a procedure and an edge $e \in E$ corresponds to a procedure call. For all $n \in N$, $\mathcal{L}_N(n)$ is the entry block of the procedure corresponding to n. For all $e \in E$ such that $e = (n, n')$, $\mathcal{L}_E(e)$ is b^e for some label b containing a call from the procedure corresponding to n to the procedure corresponding to n'.

For a DAG $\mathcal{D} = (r, N, E, \mathcal{L}_N, \mathcal{L}_E)$, and a tuple (n_1, b, n_2) such that $n_1, n_2 \in N$ and $b \in Label$, define $\mathcal{D} + (n_1, b, n_2)$ as the DAG obtained by adding an edge from n_1 to n_2, i.e., $(r, N, E \cup \{(n_1, n_2)\}, \mathcal{L}_N, \mathcal{L}_E \cup \{(n_1, n_2) \mapsto b^e\})$. The condition in Fig. 10 checks any node n can be picked in line 20 as long as the DAG obtained by adding $(Src[c], CallSite[c], n)$ is *consistent*. For any node $n \in N$, let $paths(n)$ be the set of all paths from root r to n. This set is guaranteed to be finite because $\mathcal{D}$ is acyclic. Given a path $p = (e_1, e_2, \cdots, e_n)$, let $conf(p)$ be the configuration $(\mathcal{L}_E(e_n) \cdot \mathcal{L}_E(e_{n-1}) \cdots \mathcal{L}_E(e_1))$. Given $n \in N$, let $conf(n) = \{\mathcal{L}_N(n) \cdot conf(p) \mid p \in paths(n)\}$, the set of all dynamic procedure instances represented by n.

DEFINITION 2. *A* DAG $\mathcal{D} = (r, N, E, \mathcal{L}_N, \mathcal{L}_E)$ *is* consistent, *denoted* $IsConsistent(\mathcal{D})$, *if* $Disj(c_1, c_2)$ *holds for each* $n \in N$ *and distinct configurations* $c_1, c_2 \in conf(n)$.

In a consistent DAG, each node can only represent a set of mutually-disjoint configurations. Clearly, when a DAG $\mathcal{D}$ is a tree, i.e., $paths(n)$ is a singleton set for each n, then $\mathcal{D}$ is consistent. It is easy to argue, given the particular resolution of line 20 in Fig. 10, that $IsConsistent(CreateDAG())$ is a loop invariant of Fig. 8 at line 18.

Fix a particular execution of Gen_VC for the rest of the section. Let gs be the set of global variables of the input program. Let $\mathcal{D} = (r, N, E, \mathcal{L}_N, \mathcal{L}_E)$ be the result of calling $CreateDAG()$ after Gen_VC finishes. For any node $n \in N$, let $Proc(n)$ be the procedure represented by n. Let $\mathcal{D}_n$ be the sub-DAG of $\mathcal{D}$ rooted at node n. Let $N_n \subseteq N$ and $E_n \subseteq E$ be the set of nodes and edges of $\mathcal{D}_n$, respectively. Let F be a function from N to the set of formulas such that $F(n)$ is the following:

$$\begin{array}{ll} & Control[n] \wedge pVC(n) \\ \bigwedge_{m \in N_n \setminus \{n\}} & (Control[m] \Rightarrow pVC(m)) \\ \bigwedge_{e \in E_n} & (Control[e] \Rightarrow Control[Dest[e]] \\ & \qquad \wedge In[e] = In[Dest[e]] \\ & \qquad \wedge Out[e] = Out[Dest[e]]) \end{array}$$

where each of $Control$, $Dest$, In, Out are the corresponding global variables of Fig. 8 at the end of Gen_VC's execution. Further, $pVC(n)$ is the procedural VC created by Gen_pVC (by a sequence of calls to $Push$) on the unique invocation that returned node n. Let $\mathcal{C}(\varphi)$ be the set of symbolic constants of the formula φ. It is easy to establish the following results:

- The formula generated by Gen_VC is syntactically equivalent to $F(r)$.
- For any two distinct nodes $n_1, n_2 \in N$, $\mathcal{C}(pVC(n_1)) \cap \mathcal{C}(pVC(n_2)) = \emptyset$. Further, if $N_{n_1} \cap N_{n_2} = \emptyset$, then $\mathcal{C}(F(n_1)) \cap \mathcal{C}(F(n_2)) = \emptyset$.
- For any edge $e = (n_1, n_2) \in E$, $(F(n_1) \wedge Control[n_2]) \Rightarrow F(n_2)$.

The following theorem immediately establishes the correctness of Gen_VC. The rest of this section is devoted to proving this theorem.

THEOREM 1. *The formula* $F(r)$ *is satisfiable if and only if* $Proc(r)$ *has a terminating execution.*

We first mention some results on pVC generation (without proof). For $n \in N$, let $\{e_1, e_2, \cdots, e_k\}$ be the set of out-going edges of n such that the edge e_i represents the i^{th} procedure call in $Proc(n)$ (for some arbitrary numbering of procedure calls in

$Proc(n)$). Let $Proc'(n)$ be a procedure that is the same as $Proc(n)$ except that the i^{th} procedure call is replaced with *havoc gs* followed by $assume\ Control[e_i]$. (Thus, $Proc'(n)$ is call-free.) The *havoc* over-approximates the effect of the procedure call, and the control constant $Control[e_i]$ tracks if the i^{th} call is taken (without actually going into the callee).

Let $Value$ be a type referring to the set of *values* over which program expressions are evaluated. Let *state* refer to a map of type $[Global]\,Value$, denoting a valuation of the global variables. Let *model* refer to a map of type $[Const]\,Value$, denoting an assignment to symbolic constants. Given a model $\mathcal{M}$ and a map m of type $[Global]\,Const$, let $\mathcal{M}(m)$ be the state obtained as the composition $\mathcal{M} \circ m$.

For convenience, we say that an execution τ of $Proc'(n)$ *takes* edge e if e is an out-going edge of n and τ goes through $assume\ Control[e]$. Further, in this case, the *actual-in* (respectively, *actual-out*) state of τ at e is the state before (respectively, after) the corresponding $havoc$.

The following results hold of pVC generation:

- If $pVC(n)$ is satisfiable, say with a model $\mathcal{M}$, then there exists an execution τ of $Proc'(n)$ starting in state $\mathcal{M}(In[n])$ and ending in state $\mathcal{M}(Out[n])$. If τ takes edge e then $\mathcal{M}(Control[e])$ is *true* and states $\mathcal{M}(In[e])$ and $\mathcal{M}(Out[e])$) are the actual-in and actual-out states, respectively, of τ at e.
- If there is an execution τ of $Proc'(n)$ then $pVC(n)$ is satisfiable with a model $\mathcal{M}$ such that $\mathcal{M}(In[n])$ is the input state of τ, $\mathcal{M}(Out[n])$ is the output state of τ, $\mathcal{M}(Control[e])$ is *true* if τ takes edge e, and $\mathcal{M}(In[e])$ and $\mathcal{M}(Out[e])$ are the actual-in and actual-out states, respectively, of τ at e.

We now prove (a stronger version of) one side of Thm. 1: given a satisfying model $\mathcal{M}$ of $F(n)$, there is an execution of $Proc(n)$ that goes from state $\mathcal{M}(In[n])$ to $\mathcal{M}(Out[n])$.

We proceed by induction on the size of N_n (the number of descendant nodes of n). If the size of this set is 1, then this claim trivially follows from the results on pVC generation. Otherwise, because $\mathcal{M}$ satisfies $pVC(n)$, there must be an execution τ_n of $Proc'(n)$ that goes from state $\mathcal{M}(In[n])$ to $\mathcal{M}(Out[n])$. Further, if e is an out-going edge of n and τ_n takes e then actual-in and actual-out states of τ at e are $\mathcal{M}(In[e])$ and $\mathcal{M}(Out[e])$. We will extend τ_n to a full execution of $Proc(n)$.

If edge $e = (n, m)$ is taken by τ_n then $\mathcal{M}(Control[e])$ is *true*, hence $\mathcal{M}(Control[m])$ must be *true* as well. Because $\mathcal{M}$ satisfies $(F(n) \wedge Control[m])$, it must satisfy $F(m)$. By induction hypothesis (on m), we get an execution τ_m of $Proc(m)$ that goes from state $\mathcal{M}(In[m]$ to $\mathcal{M}(Out[m])$. This execution can be stitched inside τ_n to replace the skipped call at e because $\mathcal{M}(In[e]) = \mathcal{M}(In[m])$ and $\mathcal{M}(Out[e]) = \mathcal{M}(Out[m])$. We can repeat this for all edges taken by τ_n to obtain a full execution of $Proc(n)$. This completes this side of the proof. Note that we did not require $\mathcal{D}$ to be consistent; this side of the proof holds for any DAG.

We now prove (a stronger version of) the second side of Thm. 1: given an execution τ of $Proc(n)$ that goes from state v_{in} to v_{out}, there is a satisfying model $\mathcal{M}$ of $F(n)$ such that $v_{\text{in}} = \mathcal{M}(In[n])$ and $v_{\text{out}} = \mathcal{M}(Out[n])$.

The proof is by induction on the size of N_n. Again, if the size of this set is 1 then the result follows from pVC generation. Otherwise, let τ_n be a sub-execution of τ that corresponds to an execution $Proc'(n)$. (τ_n can be obtained simply by taking out the callee traces of $Proc(n)$ from τ.) For an edge e taken by τ_n, let v^e_{in} and v^e_{out} be the actual-in and actual-out states, respectively, of τ_n on e. By properties of pVC generation, there is a model $\mathcal{M}_n$ of $pVC(n)$ such that $v_{\text{in}} = \mathcal{M}_n(In[n])$ and $v_{\text{out}} = \mathcal{M}_n(Out[n])$. Further, $v^e_{\text{in}} = \mathcal{M}_n(In[e])$ and $v^e_{\text{out}} = \mathcal{M}_n(Out[e])$.

Let T_n (resp., U_n) be the set of out-going edges of n that are taken (resp., not taken) by τ. For $e = (n, m) \in T_n$, let τ_e be the sub-execution of τ for $Proc(m)$. τ_e must go from state v^e_{in} to state v^e_{out}. By induction hypothesis (on m), there is a model $\mathcal{M}_e$ of $F(m)$ such that $\mathcal{M}_e(In[m]) = v^e_{\text{in}}$ and $\mathcal{M}_e(In[m]) = v^e_{\text{out}}$. For two distinct edges $a, b \in T_n$, we know that $\mathcal{L}_E(a)$ and $\mathcal{L}_E(b)$ must be non-disjoint because they are taken on the same execution. Thus, it must be that $N_{Dest[a]} \cap N_{Dest[b]} = \emptyset$ because otherwise $Dest[a]$ and $Dest[b]$ will have a common descendant, violating consistency of $\mathcal{D}$ (extensions of non-disjoint configurations are non-disjoint). Therefore, $\mathcal{C}(F(Dest[a])) \cap \mathcal{C}(F(Dest[b])) = \emptyset$. Define $\mathcal{M} = \mathcal{M}_n \cup_{e \in T_n} (\mathcal{M}_e \cup (Control[e] \mapsto \textit{true})) \cup_{e \in Q_n} (Control[e] \mapsto \textit{false})$. This union is well-defined because the models $\mathcal{M}_e$ for $e \in T_n$ are over disjoint sets of constants. It is easy to show that $\mathcal{M}$ satisfies $F(n)$, completing the proof.

3.3 Checking DAG Consistency

Defn. 2 does not give an efficient way of deciding if a DAG is consistent because: (1) deciding disjointness of two configurations involves transitive reachability queries, and (2) a DAG may represent an exponential number of configurations. Lem. 1 helps address the first problem. Alg. 1 exploits Lem. 1 to address the second problem.

For any $b \in Label$, let $Blk(b) = Blk(b^e) = b$. For two blocks $\gamma_1, \gamma_2 \in \Gamma$, let $Disj_{blk}(\gamma_1, \gamma_2)$ hold if and only if there is no path in the control flow graph of the procedure from $Blk(\gamma_1)$ to $Blk(\gamma_2)$ or vice versa.

LEMMA 1. *Let $c_1 = u\gamma_1 w$ and $c_2 = v\gamma_2 w$ be two distinct configurations such that $\gamma_1 \neq \gamma_2$. Then $Disj(c_1, c_2)$ holds if $Disj_{blk}(\gamma_1, \gamma_2)$ holds.*

Proof [sketch]. First we show that for any configuration c, it is never the case that $c \rightsquigarrow^+ c$, by induction on the length of c. For configurations of length 1, this simply reduces to intraprocedural reachability and we know that control-flow graphs of procedures are acyclic. For the inductive case, let $c = \gamma c'$. There are two options when we start following the transition relation from c. The first option is that the top element γ is popped off (after one or more steps) to reach c'. Applying the induction hypothesis to c', once we are at c' we can never get back to c', which implies we cannot get to c. The second option is that the top element is not popped. Then again the problem reduces to intraprocedural reachability.

Next, for $c_1 = u\gamma_1 w$ to get to $c_2 = v\gamma_2 w$ we know that it must first pop off the unmatched prefix and get to $\gamma_1 w$. Because $Disj_{blk}(\gamma_1, \gamma_2)$ holds, we know that $\gamma_1 w$ cannot get to $\gamma_2 w$ unless γ_1 is popped off. If γ_1 is popped off, we get to w but then, if we take any step, we can never get back to w. Thus, we cannot get to c_2.□

Lem. 1 implies that computation of $Disj_{blk}(.,.)$ is enough to decide disjointness of configurations. $Disj_{blk}(.,.)$ is easily computed using a topological sort of the (acyclic) control-flow graph of each procedure in isolation. This computation requires time quadratic in size of the control-flow graph of a procedure and linear in the number of procedures. After this preprocessing, deciding disjointness of two configurations simply requires identifying the longest common suffix and consulting the $Disj_{blk}(.,.)$ table once. As an example, consider the DAG shown in Fig. 11. A path from the root to a node in the graph represents a configuration. Let us pick two configurations $c_1 = \gamma_{10}\gamma_8\gamma_2$ and $c_2 = \gamma_{11}\gamma_9\gamma_2$ terminating in node n_8. By Lem. 1, $Disj(c_1, c_2)$ holds if $Disj_{blk}(\gamma_8, \gamma_9)$ holds. This is the same as following the paths that generated c_1 and c_2 starting from the root, finding the first node where they diverge (which is n_3) and then deciding disjointness of the labels of the next edges on the paths.

For verifying consistency of the entire DAG, a strawman algorithm could repeat the above process for all configurations rep-

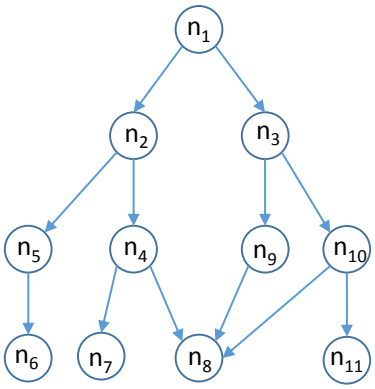

Figure 11. A DAG

Algorithm 1 Deciding consistency

Input: DAG $\mathcal{D} = (r, N, E, \mathcal{L}_N, \mathcal{L}_E)$
Output: *true* if and only if $\mathcal{D}$ is consistent

```
1: for all n ∈ N do
2:     for all n1, n2 ∈ Succ(n), n1 ≠ n2 do
3:         if Desc(n1) ∩ Desc(n2) = ∅ then
4:             continue
5:         end if
6:         if ¬Disj(L_E(n, n1), L_E(n, n2)) then
7:             return false
8:         end if
9:     end for
10: end for
11: return true
```

resented by each node, leading to exponential complexity. Alg. 1 avoids this blowup by making the observation that that a local disjointness check $Disj_{blk}(.,.)$ for two edges (n, x) and (n, y) coming out of a node n contributes to the disjointness checking for every node m that is reachable from both x and y. For example, it suffices to check only $Disj_{blk}(\gamma_1, \gamma_2)$ and $Disj_{blk}(\gamma_8, \gamma_9)$ to conclude that the DAG in Fig. 11 is consistent. Alg. 1 captures this intuition. In this algorithm, $Succ(n)$ is the set of all successor nodes of n and $Desc(n)$ is the set of all descendant nodes of n. The algorithm iterates over each pair of edges from some node to two distinct successor nodes. A $Disj_{blk}(.,.)$ check is performed if the two successor nodes have a common descendant. It is easy to see that Alg. 1 has quadratic complexity in the size of the input DAG, once the table for $Disj_{blk}(.,.)$ has been pre-computed.

THEOREM 2. *If Alg. 1 returns true then $IsConsistent(\mathcal{D})$ holds.*

We note that both Alg. 1 and Lem. 1 are, in fact, precise. That is: (1) Alg. 1 returns *true* only if $IsConsistent(\mathcal{D})$ holds, and (2) $Disj(u\gamma_1 c, v\gamma_2 c)$ holds only if $Disj_{blk}(Blk(\gamma_1), Blk(\gamma_2))$ holds. The reason is that disjointness is only defined over the control structure of the program (and also the fact that in our programming language procedure executions must return, i.e., for any $\gamma \in \Gamma$, $\gamma c \rightsquigarrow^* c$). The control structure is only an abstraction of the program. Stronger version of disjointness between configurations are possible; it is only required that no program execution goes through both configurations. We leave the benefit of stronger definitions, which can classify more configurations as disjoint, but possibly at a larger analysis cost, for future work.

3.4 Merging Strategies

It is possible that there are many compatible nodes that can be added to the DAG so that it remains consistent. In this section, we discuss strategies for picking a node among various correct choices. Let $\mathcal{D}$ be a DAG, n a node of $\mathcal{D}$ and γ a label representing an open call out of node n. Let M be the set of all nodes m such that $IsConsistent(\mathcal{D} + (n, \gamma, m))$ holds. In each strategy, if M is empty, we return $None$. Otherwise, we have the following options.

FIRST: List M in chronological order, i.e., in the order in which they were added to $\mathcal{D}$, and return its first element.

RANDOM: With low probability, return $None$ (even if M is non-empty), otherwise return a randomly picked element of M.

RANDOMPICK: Return a randomly picked element of M.

MAXC: Return the element of M that has the largest number of descendants in $\mathcal{D}$. (The intuition is that the bigger the sub-dag rooted at a node, the more the sharing when merged with that node.)

OPT: All of the previous strategies are greedy; they base their decision solely on the current DAG. In the OPT strategy, we first precompute a DAG $\mathcal{D}_o$ that represents the best possible compression of the fully-inlined tree of the input program. OPT maintains the invariant that $\mathcal{D}$ is always embedded inside $\mathcal{D}_o$. When asked for a node to merge against, OPT looks for an edge (n, m) in $\mathcal{D}_o$ with label γ. If it exists, then it returns m, otherwise it returns $None$. This strategy guarantees that as $\mathcal{D}$ grows, it remains embedded inside $\mathcal{D}_o$ (thus, $\mathcal{D}$ can never become larger than $\mathcal{D}_o$).

The DAG $\mathcal{D}_o$ is computed as follows. In the fully inlined tree, for each procedure, we consider the set of all dynamic instances of that procedure and construct their induced conflict graph (see Section 2). Next, we color this graph with minimum colors possible, and merge all nodes with the same color. In our experiments, when all calls needed to be inlined, OPT performs the best, followed by FIRST. However, OPT is too expensive because the computation of $\mathcal{D}_o$ requires reasoning over all dynamic instances. Further, when only a part of the program needs to be inlined, OPT need not be optimal.

Our default choice for a merging strategy is FIRST. It is very fast in practice (with overhead less than 0.4%) yet provides compression close to OPT in the limit.

4. Evaluation

We evaluate DAG inlining on several real-world verification instances obtained from the Static Driver Verifier (SDV) [16]. SDV is a commercial tool offered by Microsoft to third-party driver developers. SDV comes packaged with several *rules* (or properties) that a driver must satisfy. A developer can ask SDV to verify if the driver violates any of these rules. Internally, SDV compiles the driver and instruments the rule to end up with a verification instance (a program with assertions) and feeds it to a verifier. SDV currently uses Corral as the verification engine [12].

Corral takes a bound R as input and unrolls loops up to R iterations and unfolds recursion up to R recursive calls to produce a hierarchical program. (This expansion is not done up-front, but lazily.) Next, it feeds the hierarchical program to an algorithm called stratified inlining (SI) to decide reachability. SI inlines procedures lazily; it can be thought of a particular instance of Fig. 8 that intelligently implements "pick" to decide which procedure to inline next. However, it still unfolds to a tree. SI can stop early when it determines that enough of the program has been inlined to conclude that it is safe. We implemented DAG inlining (DI) using the framework of SI, with merging strategy FIRST used by default.

Our benchmark suite consists of 105 drivers and 180 rules. The size of these drivers range from 2KLOC to 40KLOC, and the total lines of code across all drivers exceeds 800KLOC. The number of procedures per driver ranges from 106 to 531. Of all verification

Algorithm	#TO	#Bugs	#Inlined (avg.)	Time (1000 s) Bug	No-bug
SI-Inv	418	51	885.2	9.1	50.7
DI-Inv	354	64	271.6	5.4	25.2
SI+Inv	358	72	759.3	13.6	82.6
DI+Inv	280	83	272.4	9.3	44.3

Figure 12. Results, in aggregate, for the SDV benchmarks. #TO refers to timeouts (2000 seconds) and spaceouts (8 GB), #Bugs is the number of bugs reported (at most one per instance), #Inlined is the number of procedures inlined before reaching a verdict, and Time is reported in thousands of seconds, divided into instances that reported a bug and ones that finished without reporting a bug.

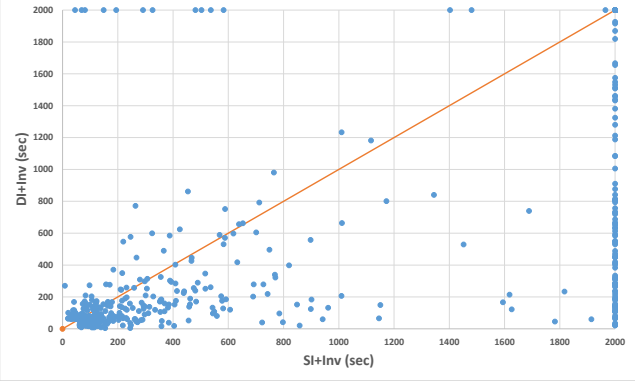

Figure 13. Scatter plot of SI+Inv vs. DI+Inv

instances generated by SDV, we only took ones on which either SI or DI took at least 60 seconds to finish. There were 619 such instances.

Corral uses invariant generation techniques as pre-pass [14]. Any inferred invariant is injected into the program as an assume statement. Invariants can be a powerful mechanism to prune search. In the limit, when the invariants are strong enough to prove the assertion correct, the search can conclude trivially. We compare SI and DI on the 619 SDV verification instances, both when they had invariants inserted and when they did not have them inserted. We use the terms "SI+Inv" and "SI-Inv" to refer to running SI on files with invariants inserted and without invariants inserted, respectively. Similarly for "DI+Inv" and "DI-Inv".

Fig. 12 presents the aggregate results over these benchmarks. For example, all instances on which SI-Inv returned a bug took a total of 9100 seconds. There were 51 such instances (#Bugs). SI-Inv timed out on 418 instances (#TO), and inlined 885.2 procedures per instance (on average) when it finished. Because we had focussed only on the hard verification instances, the large proportion of timeouts is not surprising. Inlining 885.2 procedures on average is already a small number compared to full tree inlining that can easily inline a million procedure instances (see Fig. 4). The main reason for reduction is the property-guided pruning of SI. Further, previous work shows that Corral with SI already outperforms other verifiers (e.g., SLAM [3]) on these benchmarks [12]. This justifies why these benchmarks are good for evaluating further improvements by using DAG inlining.

Fig. 12 shows that DI results in a reduction in the number of timeouts, finds more bugs, inlines fewer procedures, and improves running time for both buggy and non-buggy instances. In particular, in the absence of invariant injection (SI-Inv vs. DI-Inv), there is a 15.3% reduction in timeouts, 25.5% increase in the number of

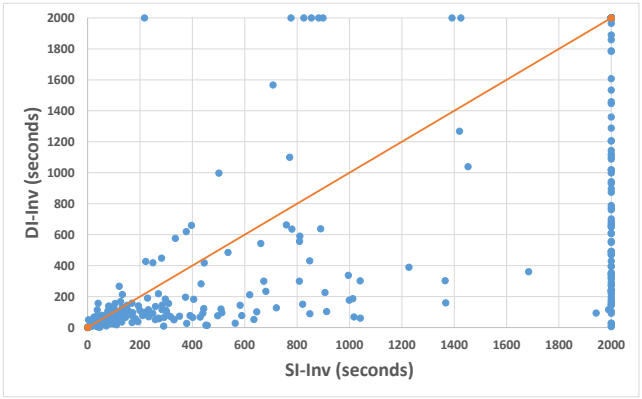

Figure 14. Scatter plot of SI-Inv vs. DI-Inv

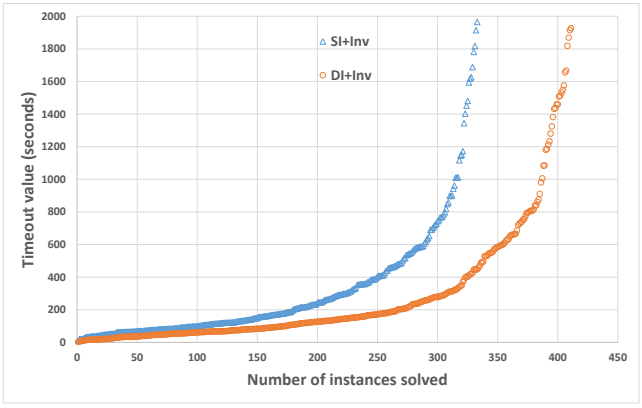

Figure 15. Cactus plot of SI+Inv vs. DI+Inv

bugs found, 3.2X reduction in number of procedures inlined and 1.95X reduction in the running time. In the presence of invariants, the comparison is not much different: there is 21.8% reduction in timeouts, 15.2% increase in the number of bugs found, 2.78X reduction in the number of procedures inlined and 1.8X reduction in time.

It may be confusing to see that in the presence of invariants, the running time (for both SI and DI) is higher. However, note that we are reporting the running time for only those instances that did not timeout or spaceout. More instances were solved when using invariants, hence the higher cumulative running time. We also note that whenever any of the two techniques returned an answer for a verification instance, it was the same answer, giving us confidence that our implementation does not have bugs.

We also present two other visualizations of the comparisons. Fig. 13 and Fig. 14 show a scatter plot of the running time of SI and DI with and without the use of invariants, respectively. While DI inlines fewer procedures than SI, this did not always result in a speedup.[1] Nonetheless, the overall trend is positive. In fact, DI+Inv was an order of magnitude faster than SI+Inv on 5% of the instances on which they both finished. DI+Inv was 5X faster on 14% of the instances.

[1] Ultimately, every technique (including ones used inside SMT solvers, not just DAG inlining) is a heuristic because of the intractable nature of the problem (Section 5), making it difficult to guarantee speedups across the board.

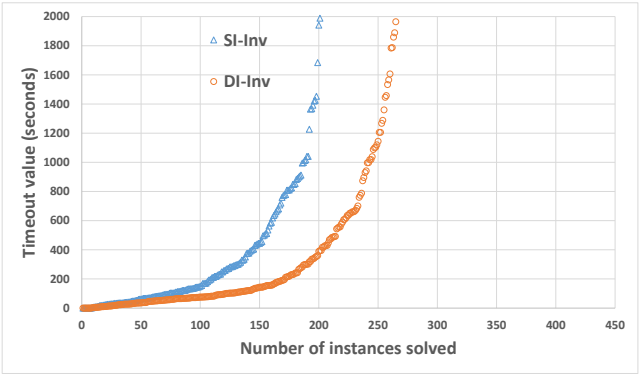

Figure 16. Cactus plot of SI-Inv vs. DI-Inv

Tree	OPT	FIRST	MAXC	RANDOM	RANDOMPICK
312231	1582	1673	1629	5096	1760
348329	16223	16261	16263	18539	16256
486713	9379	9751	9960	15406	10376
499799	4854	6400	6920	T/O	10191
553790	2793	2846	2848	8555	2860
621285	17182	22589	22545	T/O	24921
964394	1796	1890	1878	5258	2231
1125448	21459	22068	21997	42743	22733
1177613	17419	17576	17557	30873	18463
1384747	8315	8527	8484	T/O	10268
1390941	4887	4999	5109	13327	5306
2645020	8623	8798	9017	18246	9270
3211353	T/O	13782	13837	18976	13847
Dev:	–	8%	9%	129%	21%

Figure 17. A comparison of the number of procedures inlined by different merging strategies when all dynamic instances must be inlined. T/O implies that the fully-inlined DAG could not be created within 900 seconds.

Fig. 15 and Fig. 16 show a cactus plot of SI against DI, with and without the use of invariants, respectively. The X-axis is the number of instances solved and Y-axis is the time taken. Again, the trends with and without invariants are similar, except that more instances are solved by SI and DI when using invariants. The figures show that DI solves more instances than SI irrespective of the timeout value chosen.

Merging strategies In the context of lazy inlining that may not need to inline all procedures, it is hard to design an optimal merging technique because the tree that needs to be inlined is not known up front.

In order to evaluate the performance of a merging strategy, we keep inlining until all dynamic instances get inlined. Then we compared the number of inlined procedures against our optimal strategy and measured the difference between them.

Fig. 17 shows the number of procedures inlined by different merging strategies on a small subset of the SDV benchmarks. The first column (Tree) gives the size of fully inlined tree and the rest of the columns give the DAG sizes under different strategies. For both RANDOM and RANDOMPICK, we did five runs and took their average.

The last row of Fig. 17 shows the deviation of various merging strategies against OPT. As we can see, FIRST performs the best with just 8% deviation from the optimal, on average. RANDOM performs the worst, indicating that it is best to take the opportunity for merging when there is one.

We also measured the total time spent inside the routine that looks for a candidate to merge each time a procedure needs to be inlined. (This time is included in the time taken by DI.) It turns out to be insignificant; it is 0.4% of the total time taken by DI. This implies that one can invest in more aggressive merging techniques without adding an overhead. For instance, we can decide disjointness by reasoning over data values as well, but we leave this for future work.

5. Discussion

Asymptotic complexity For a programming language whose operational semantics can be encoded in SMT, deciding reachability in single-procedure hierarchical programs is in NP, but NEXPTIME-hard for general hierarchical programs [11]. This jump in complexity shows up in practical algorithms, namely ones implemented in BMC tools. These algorithms suffer from a doubly-exponential complexity: one exponent in generating the SMT formula (after inlining) and another that is incurred inside the SMT solver for finding a satisfying model.

The proof of NEXPTIME-hardness relies crucially on the program generating exponentially long paths. We know that the complexity is lower (in NP) when the program only generates polynomially sized paths. The program in Fig. 2 is one example that demonstrates the lower complexity. However, we are unaware of any efficient VC generation that exploits shorter path sizes.

DAG inlining is our first step towards reducing the exponential explosion in VC generation. DAG inlining by itself does not fundamentally reduce asymptotic complexity because it relies on disjointness, which a program may not have. However, in practice, it does provide good compression, leading to performance improvements. As future work, we aim to study the general problem of VC generation for hierarchical programs with at most polynomially long paths.

Procedure inlining Procedure inlining is a well-understood concept that comes from compiler optimizations. Many program analysis tools borrow this technology and performing inlining at the source level. For example, SMACK [17] uses procedure inlining that is provided by its LLVM frontend. DAG inlining, however, is hard to perform at the source level. Our intuition behind DAG inlining was enabled by performing inlining at the VC level (i.e., at a logical formula level).

Inlining at the VC level has its own cost. For instance, CBMC performs inlining at the C level and then uses a pointer analysis to compile away pointers. The pointer analysis can be very precise after procedures have been inlined. In fact, many pointer accesses can get resolved exactly because procedures are analyzed with their complete calling context. If one were to use DAG inlining inside CBMC, one would need to run pointer analysis without inlining. Thus, one would have to offset the benefit of DAG inlining against the precision lost inside the pointer analysis. We leave this investigation as future work.

6. Related Work

Merging paths in symbolic execution The idea of merging states symbolically has been used for improving the efficiency of path-based symbolic execution [10, 19]. The main idea is that if two symbolic paths (i.e., executions where some data values are represented symbolically) are similar enough, then their symbolic states are merged together. Then exploration discards the original paths and continues with the new merged path. This can improve efficiency because continuing the merged path does the work of continuing the multiple constituent paths. However, this is very different from our use of merging. For instance, we merge on the call tree, not on program executions. Further, our merging is for disjoint executions, whereas in symbolic execution, similar (perhaps overlapping) paths are what get merged.

Lazy inlining The problem of lazily inlining procedures, where the full unrolling of the call graph as a tree is avoided, has been studied in different forms. Stratified inlining [14], *structural abstraction* [1], *inertial refinement* [18] and *scope bounding* [9, 15] all explore lazy inlining. However, in each case, the unfolded call graph is still a tree and there is no merging. Our algorithm integrates nicely with lazy inlining, as shown in Fig. 8.

Procedure summarization A popular way of avoiding inlining of procedures is to *summarize* them and reuse the summaries in different calling contexts. Procedure summarization can help avoid the cost of inlining. While summarization is an effective optimization (and should be used whenever applicable), it does not fundamentally change the complexity of deciding reachability in hierarchical programs over arbitrary (decidable) theories.

MOPED [8] and BEBOP [2] are summarization-based tools for analyzing (recursive) Boolean programs. They offer much better worst-case complexity than full inlining of procedures (even up to a bound). However, they are fundamentally restricted to programs with variables over a finite domain. Their summarization techniques are not guaranteed to terminate when, say, the program operates over mathematical integers. Thus, such tools cannot be applied directly while dealing with hierarchical programs over SMT theories like linear arithmetic, arrays, etc. However, they are used indirectly inside software model checking tools like SLAM [3] where programs are abstracted to a Boolean program and then analyzed using summarization. Comparisons to SLAM-like techniques are orthogonal to the goals of this paper, and have been discussed elsewhere [12, 14].

7. Conclusions

We have presented a method called DAG inlining as an alternative to the standard practice of doing tree-based procedure inlining for deciding reachability in hierarchical programs. DAG inlining is a way of compressing the VC of a hierarchical program. It relies on the notion of disjoint procedures that cannot be taken on the same program execution. We empirically evaluated that there is enough opportunity to compress in real world programs and it leads to significant performance improvements. We also evaluated different merging strategies that can result in different amounts of compressions.

References

[1] D. Babic and A. J. Hu. Calysto: Scalable and precise extended static checking. In *International Conference on Software Engineering*, pages 211–220, 2008.

[2] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN*, pages 113–130, 2000.

[3] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation*, 2001.

[4] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Program Analysis for Software Tools and Engineering*, 2005.

[5] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, pages 364–387, 2005.

[6] CBMC. Bounded Model Checking for ANSI-C. `http://www.cprover.org/cbmc/`.

[7] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.

[8] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *Computer Aided Verification*, pages 324–336, 2001.

[9] F. Ivancic, G. Balakrishnan, A. Gupta, S. Sankaranarayanan, N. Maeda, H. Tokuoka, T. Imoto, and Y. Miyazaki. DC2: A framework for scalable, scope-bounded software verification. In *Automated Software Engineering*, 2011.

[10] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *Programming Language Design and Implementation*, 2012.

[11] A. Lal and S. Qadeer. Reachability modulo theories. In *Reachability Problems*, 2013.

[12] A. Lal and S. Qadeer. Powering the Static Driver Verifier using Corral. In *Foundations of Software Engineering*, 2014.

[13] A. Lal and S. Qadeer. A program transformation for faster goal-directed search. In *Formal Methods in Computer Aided Design*, 2014.

[14] A. Lal, S. Qadeer, and S. Lahiri. Corral: A solver for reachability modulo theories. In *Computer Aided Verification*, 2012.

[15] A. Loginov, E. Yahav, S. Chandra, S. Fink, N. Rinetzky, and M. G. Nanda. Verifying dereference safety via expanding-scope analysis. In *International Symposium on Software Testing and Analysis*, 2008.

[16] Microsoft. Static driver verifier. `http://msdn.microsoft.com/en-us/library/windows/hardware/ff552808(v=vs.85).aspx`.

[17] Z. Rakamaric and M. Emmi. SMACK: Decoupling source language details from verifier implementations. In *Computer Aided Verification*, 2014.

[18] N. Sinha. Modular bug detection with inertial refinement. In *Formal Methods in Computer Aided Design*, 2010.

[19] E. Torlak and R. Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *Programming Language Design and Implementation*, 2014.