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

In: Programming Language Design and Implementation (PLDI) 2015

Akash Lal and Shaz Qadeer
Microsoft Research
Bug Finding via Bounded Verification

• Several success stories of automated verification
  • Static Driver Verifier, F-Soft, Facebook Infer, ...
  • In finding bugs!

• Design for finding bugs quickly
  • Instead of discovering them as a by-product of proof failure

• Bounded verification: analyze a (useful) subset of program behaviors really fast
Bounded Verification

- Build efficient decision procedures for Bounded Verification
  - Inspired by the success in Hardware verification

<table>
<thead>
<tr>
<th>SV-COMP 2015</th>
<th>ALF</th>
<th>DEALINGS</th>
<th>HP</th>
<th>HKX</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Verifier</strong></td>
<td>CEGAR</td>
<td>Predicate Abstraction</td>
<td>Symbolic Execution</td>
<td>Bounded Model Checking</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>k-Induction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Property-Directed Reachability</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Explicit-Value Analysis</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Numerical Interval Analysis</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ARC-Based Analysis</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Lazy Abstraction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Interpolation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Automata-Based Analysis</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Concurrency Support</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Ranking Functions</td>
</tr>
</tbody>
</table>

Static Driver Verifier uses Corral [FSE’14]
Bounded Verification done previously

- Standard approach: Inline all procedures, generate a SAT/SMT constraint, invoke solver

- Procedure Inlining causes exponential blowup
Our Work: DAG Inlining

**Program Call Graph**

**Standard Procedure Inlining**

**DAG Inlining**

- Exponential Blowup in Program Size
- Potentially Exponential Savings!

2^i copies of P_i

DAG structure program dependent
Micro Benchmark

var g: int;

procedure main() {
    g := 0;
    if(*) { call P_0(); } 
    else { call P_0(); }
}

procedure P_i() {
    g := g + 1;
    if(*) { call P_{i+1}(); } 
    else { call P_{i+1}(); }
}

procedure P_N() {
    assert g == N;
}
Talk Outline

• Problem definition
• VC Generation Algorithms [Background]
  • Single-procedure programs
  • Multi-procedure programs
• DAG Inlining Algorithm
• Evaluation
Definitions

- **Hierarchical Programs**: Sequential programs without loops and recursion

- **Reachability Modulo Theories (RMT)**: Find a terminating execution of a program whose operational semantics can be encoded in decidable logic.
Programming Language

// variables
var x: T

// functions
func f: T \rightarrow T

// commands
x := expr
assume expr
call foo(x);

// procedures
procedure foo(args) {
    LocalDecls;
    Body;
}

// control
if(expr) { cmds; } else { cmds; }
while(*) { cmds; }

Example:
Linear arithmetic via the theory of linear arithmetic (LIA)
Non-linear arithmetic via uninterpreted functions (EUF)
Memory lookup using theory of arrays
Floating-point as uninterpreted
Reachability Modulo Theories

• Finding assertion failures is equivalent to finding terminating executions

```
assert e;
err := e;
if(!err) return;

call foo();
call foo();
if(!err) return;

main() {
    ...
    return;
}
main() {
    err := true;
    ...
    assume !err;
    return;
}
```
Reachability Modulo Theories

• RMT in hierarchical programs is *decidable* (*NEXPTIME hard*) [RP’13]
  • Can’t hide all sorts of complexity behind undecidability

• Direct application to bounded verification
  • E.g., “Bounded Model Checking”

• Relevant to unbounded verification
  • Checking inductive proofs (loop invariants) without pre-post conditions
Talk Outline

• Problem definition
• VC Generation Algorithms ↔ Solving RMT in Hierarchical programs
  • Single-procedure programs
  • Multi-procedure programs
• DAG Inlining Algorithm
• Evaluation
VC Generation: Single Procedure

procedure f(w: int)
  returns (x: int, y: int)
{
  start:
    x := *;
    y := x + w;
    goto 11, 12;

  11:
    x := x + 1;
    goto 13;

  12:
    x := x + 2;
    goto 13;

  13:
    assume !(x > y);
    return;
}

VC-Gen: $f \Rightarrow \phi_f(w, x, y)$

Theorem: $\phi_f(w, x, y, z)$ holds iff $f(w)$ can return $(x, y)$

Corollary: $\phi_f$ is SAT iff $f$ has a terminating execution
VC Generation

procedure f(w: int)
    returns (x: int, y: int)
{
    start:
        x := *;
        y := x + w;
        goto l1, l2;

    l1:
        x := x + 1;
        goto l3;

    l2:
        x := x + 2;
        goto l3;

    l3:
        assume !(x > y);
        return;
}

procedure f(w: int)
    returns (x: int, y: int)
{
    start:
        x1 := *;
        y1 := x1 + w;
        goto l1, l2;

    l1:
        x2 := x1 + 1;
        goto l3;

    l2:
        x3 := x1 + 2;
        goto l3;

    l3:
        x4 := \phi(x2, x3);
        assume !(x4 > y1);
        x := x4; y := y1;
        return;
}
VC Generation

procedure f(w: int)
  returns (x: int, y: int)
{
  start:
    x := *;
    y := x + w;
    goto l1, l2;

  l1:
    x := x + 1;
    goto l3;

  l2:
    x := x + 2;
    goto l3;

  l3:
    assume !(x > y);
    return;
}

procedure f(w: int)
  returns (x: int, y: int)
{
  start:
    x1 := *;
    y1 := x1 + w;
    goto l1, l2;

  l1:
    x2 := x1 + 1; x4 := x2;
    goto l3;

  l2:
    x3 := x1 + 2; x4 := x3;
    goto l3;

  l3:
    x4 := \varphi(x2, x3);
    assume !(x4 > y1);
    x := x4; y := y1;
    return;
}
VC Generation

procedure f(w: int)
  returns (x: int, y: int)
{
  start:
    x := *;
    y := x + w;
    goto l1, l2;

  l1:
    x := x + 1;
    goto l3;

  l2:
    x := x + 2;
    goto l3;

  l3:
    assume !(x > y);
    return;
}

procedure f(w: int)
  returns (x: int, y: int)
{
  start:
    assume y1 == x1 + w;
    goto l1, l2;

  l1:
    assume x2 == x1 + 1; assume x4 == x2;
    goto l3;

  l2:
    assume x3 == x1 + 2; assume x4 == x3;
    goto l3;

  l3:
    assume !(x4 > y1);
    assume x == x4; assume y == y1;
    return;
}
VC Generation

Block constraints

\[ C_{\text{start}}: \quad y_1 = x_1 + w \]

\[ C_{11}: \quad x_2 = x_1 + 1 \land x_4 = x_2 \]

\[ C_{12}: \quad x_3 = x_1 + 2 \land x_4 = x_3 \]

\[ C_{13}: \quad \neg(x_4 > y_1) \land x = x_4 \land y = y_1 \]

procedure \( f(w: \text{int}) \)

returns (x: \text{int}, y: \text{int})

{ 

start:

assume \( y_1 = x_1 + w \);
goto 11, 12;

11:

assume \( x_2 = x_1 + 1 \); assume \( x_4 = x_2 \);
goto 13;

12:

assume \( x_3 = x_1 + 2 \); assume \( x_4 = x_3 \);
goto 13;

13:

assume \! (x_4 > y_1);
assume x = x_4; assume y = y_1;
return;

}
VC Generation

Algorithm

1. Introduce Boolean constant $b_L$ for each block $L$
2. $E_L$ is $b_L == C_L$ if $L$ ends in return
   $b_L == C_L \land \lor_{m \in \text{Succ}(L)} b_m$
3. VC($f$) is
   $b_{\text{start}} \land (\land_{L \in \text{Blocks}(f)} E_L)$

Block constraints

$E_{\text{start}}: b_{\text{start}} == (y_1 == x_1 + w) \land (b_{l1} \lor b_{l2})$

$E_{l1}: b_{l1} == (x_2 == x_1 + 1 \land x_4 == x_2) \land b_{l3}$

$E_{l2}: b_{l2} == (x_3 == x_1 + 2 \land x_4 == x_3) \land b_{l3}$

$E_{l3}: b_{l3} == (\neg(x_4 > y_1) \land x == x_4 \land y == y_1)$

procedure $f(w: \text{int})$
  returns $(x: \text{int}, y: \text{int})$
{
  start:
    assume $y_1 == x_1 + w$;
    goto $l1, l2$;
  
  $l1$:
    assume $x_2 == x_1 + 1$; assume $x_4 == x_2$;
    goto $l3$;
  
  $l2$:
    assume $x_3 == x_1 + 2$; assume $x_4 == x_3$;
    goto $l3$;
  
  $l3$:
    assume $!(x_4 > y_1)$;
    assume $x == x_4$; assume $y == y_1$;
    return;
}
procedure f(v1: int, v2: int)
  returns (r: int)
{
  var c: bool;
  goto l1, l2;

l1:
  assume c;
  call r := g(v1);
  goto l3;

l2:
  assume !c;
  call r := g(v2);
  goto l3;

l3:
  return;
}

procedure g(a: int)
  returns (b: int)
{
  b := a + 1;
}
VC Generation: Multiple Procedures

procedure f(v1: int, v2: int)
  returns (r: int)
{
  var c: bool;
  goto l1, l2;

l1:
  assume c;
  call r := g(v1); assume M0;
  goto l3;

l2:
  assume !c;
  call r := g(v2); assume M1;
  goto l3;

l3:
  return;
}

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

VC(f): (c ∧ M0) ∨ (¬c ∧ M1)
VC(g): b == a + 1

Algorithm

1. Introduce Boolean constant $M_i$ for each call
### VC Generation: Multiple Procedures

```plaintext
procedure f(v1: int, v2: int)
    returns (r: int)
{
    var c: bool;
    goto l1, l2;

l1:
    assume c;
    call r := g(v1); assume M0;
    goto l3;

l2:
    assume !c;
    call r := g(v2); assume M1;
    goto l3;

l3:
    return;
}

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

#### Algorithm

1. Introduce Boolean constant $M_i$ for each call
2. Introduce Boolean constant $N_i$ for each procedure instance
3. Connect

---

$N_0$  
\[ N_0 \Rightarrow (c \land M_0) \lor (\neg c \land M_1) \]  
VC of $f$

$N_1$  
\[ N_1 \Rightarrow (N_1 \land v_1 == a_1 \land r == b_1) \]  
formals equals actuals

$N_2$  
\[ N_2 \Rightarrow (N_2 \land v_2 == a_2 \land r == b_2) \]  
formals equals actuals

$N_3$  
\[ N_3 \Rightarrow (b_1 == a_1 + 1) \]  
VC of $g$

$N_4$  
\[ N_4 \Rightarrow (b_2 == a_2 + 1) \]  
VC of $g$
VC Generation: Multiple Procedures

procedure f(v1: int, v2: int)
returns (r: int)
{
    var c: bool;
    goto l1, l2;

l1:
    assume c;
    call r := g(v1); assume M0;
    goto l3;

l2:
    assume !c;
    call r := g(v2); assume M1;
    goto l3;

l3:
    return;
}

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

Algorithm
1. Introduce Boolean constant $M_i$ for each call
2. Introduce Boolean constant $N_i$ for each procedure instance
3. Connect
DAG Inlining: Algorithm

• **Dynamic Procedure Instances**: A procedure qualified by its call stack

```plaintext
procedure main()
{
  if(...) { A: bar(); }
  else { B: baz(); }
}

procedure baz()
{
  C: foo();
}

procedure bar()
{
  D: foo();
}
```

- [A; C; foo]
- [B; D; foo]

• **Disjoint instances**: Two procedure instances that cannot be taken on the same execution
DAG Inlining: Algorithm

• *Theorem*: Any two *disjoint* instances of the same procedure can be *merged* together when inlining.
procedure f(v1: int, v2: int) 
  returns (r: int)
{
  var c: bool;
  goto l1, l2;

l1:
  assume c;
  call r := g(v1);
  goto l3;

l2:
  assume !c;
  call r := g(v2);
  goto l3;

l3:
  return;
}
DAG Inlining: Algorithm

procedure f(v1: int, v2: int)
    returns (r: int)
{
    var c: bool;
    goto l1, l2;

    l1:
        assume c;
        call r := g(v1);
        goto l3;

    l2:
        assume !c;
        call r := g(v2);
        goto l3;

    l3:
        return;
}

Standard (Tree) Inlining

$N_0 \Rightarrow \left( c \land M_0 \right) \lor \left( \neg c \land M_1 \right)$ \quad VC of f
$M_0 \Rightarrow \left( N_1 \land v_1 \right) \land \left( \neg c \land M_1 \right)$ \quad formals equals actuals
$M_1 \Rightarrow \left( N_2 \land v_2 \right) \land \left( \neg c \land M_1 \right)$ \quad formals equals actuals
$N_1 \Rightarrow \left( b_1 \right) \land (a_1 + 1) \quad VC of g$
$N_2 \Rightarrow \left( b_2 \right) \land (a_2 + 1) \quad VC of g$

DAG Inlining

$N_0 \Rightarrow \left( c \land M_0 \right) \lor \left( \neg c \land M_1 \right)$ \quad VC of f
$M_0 \Rightarrow \left( N_1 \land v_1 \right) \land \left( \neg c \land M_1 \right)$ \quad formals equals actuals
$M_1 \Rightarrow \left( N_2 \land v_2 \right) \land \left( \neg c \land M_1 \right)$ \quad formals equals actuals
$N_1 \Rightarrow \left( b_1 \right) \land (a_1 + 1) \quad VC of g$

c \Rightarrow \left( r \right) \land \left( v_1 + 1 \right) \land \neg c \Rightarrow \left( r \right) \land \left( v_2 + 1 \right)
DAG Inlining: Algorithm

Disjoint([main; bar; foo], [main, baz, foo])

DAG Consistency: For each node n, all configurations represented by n should be mutually disjoint

Algorithm: While inlining, keep merging with existing instances as long as the DAG is consistent
Optimal merging reduces to Graph Coloring

Control-Flow Graph

Conflict Graph: Edge $\Rightarrow$ not disjoint

Optimal merging requires computation of maximal independent sets, which is equivalent to graph coloring.
Implementing the Algorithm

• We can decide disjointness in linear time based on control flow

[A1; A2; A3; A4; foo]

[A1; A2; B1; B2; B3; B4; foo]

longest common prefix

These calls should not appear on the same path in the CFG of Proc(A2)

• Disjointness of procedure instances can be resolved by disjointness in a single CFG.
Implementing the Algorithm [See Paper]

• Disjointness of two configurations in linear time
• Deciding DAG consistency in quadratic time

• Greedy graph coloring (8% off the optimal)

• Overall: Less than 0.4% time spent in DAG operations
Experiments: Static Driver Verifier

• Commercial tool, ships with Windows
  • Used internally and by third-party driver developers
  • Part of Windows Driver Certification program

• Uses Corral, an RMT solver
  • Based on Tree Inlining, but:
    • Includes several optimizations over tree inlining

• Total LOC: 800K
• Total verification time: well over a month
Compression in Practice

• Tree/DAG Sizes
Compression strategy

- We use a greedy merging strategy: turns out to be around 8% off the optimal in practice

<table>
<thead>
<tr>
<th>Tree</th>
<th>OPT</th>
<th>FIRST</th>
<th>MAXC</th>
<th>RANDOM</th>
<th>RANDOMPick</th>
</tr>
</thead>
<tbody>
<tr>
<td>312231</td>
<td>1582</td>
<td>1673</td>
<td>1629</td>
<td>5096</td>
<td>1760</td>
</tr>
<tr>
<td>348329</td>
<td>1623</td>
<td>16261</td>
<td>16263</td>
<td>18539</td>
<td>16256</td>
</tr>
<tr>
<td>486713</td>
<td>9379</td>
<td>9751</td>
<td>9960</td>
<td>15406</td>
<td>10376</td>
</tr>
<tr>
<td>499799</td>
<td>4854</td>
<td>6400</td>
<td>6920</td>
<td>T/O</td>
<td>10191</td>
</tr>
<tr>
<td>553790</td>
<td>2793</td>
<td>2846</td>
<td>2848</td>
<td>8555</td>
<td>2860</td>
</tr>
<tr>
<td>621285</td>
<td>17182</td>
<td>22589</td>
<td>22545</td>
<td>T/O</td>
<td>24921</td>
</tr>
<tr>
<td>964394</td>
<td>1796</td>
<td>1890</td>
<td>1878</td>
<td>5258</td>
<td>2231</td>
</tr>
<tr>
<td>1125448</td>
<td>21459</td>
<td>22068</td>
<td>21997</td>
<td>42743</td>
<td>22733</td>
</tr>
<tr>
<td>1177613</td>
<td>17419</td>
<td>17576</td>
<td>17557</td>
<td>30873</td>
<td>18463</td>
</tr>
<tr>
<td>1384747</td>
<td>8315</td>
<td>8527</td>
<td>8484</td>
<td>T/O</td>
<td>10268</td>
</tr>
<tr>
<td>1390941</td>
<td>4887</td>
<td>4999</td>
<td>5109</td>
<td>13327</td>
<td>5306</td>
</tr>
<tr>
<td>2645020</td>
<td>8623</td>
<td>8798</td>
<td>9017</td>
<td>18246</td>
<td>9270</td>
</tr>
<tr>
<td>3211353</td>
<td>T/O</td>
<td>13782</td>
<td>13837</td>
<td>18976</td>
<td>13847</td>
</tr>
</tbody>
</table>

Dev: – 8% 9% 129% 21%
Results: Summary

- Find more bugs in less time
- Almost twice as fast as the production system

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>#TO</th>
<th>#Bugs</th>
<th>#Inlined (avg.)</th>
<th>Time (1000 s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bug</td>
<td>No-bug</td>
</tr>
<tr>
<td>Tree-1</td>
<td>418</td>
<td>51</td>
<td>885.2</td>
<td>9.1</td>
</tr>
<tr>
<td>DAG-1</td>
<td>354</td>
<td>64</td>
<td>271.6</td>
<td>5.4</td>
</tr>
<tr>
<td>Tree-2</td>
<td>358</td>
<td>72</td>
<td>759.3</td>
<td>13.6</td>
</tr>
<tr>
<td>DAG-2</td>
<td>280</td>
<td>83</td>
<td>272.4</td>
<td>9.3</td>
</tr>
</tbody>
</table>

Production Quality
Results - 1

- Number of instances: 619
- Reduction in Timeouts: 64
- 5X speedup: 35
- 5X slowdown: 2
Results - 2

- Number of instances: 619
- Reduction in Timeouts: 78
- 5X speedup: 45
- 5X slowdown: 1
Before the last slide ...
Microsoft Research India
We are hiring!

Researchers, post-docs & engineers
Systems, ML, Crypto, Theory, PL, HCI, ICT4D,…
Summary

• Reachability in Hierarchical Programs is fundamental

• Standard (Tree) inlining causes exponential blowup
  • Limits many BMC tools to small programs

• DAG inlining refines the age-old idea of procedure inlining
  • Demonstrated significant speedups of a production system