A BDD-based Model Checker for Programs with Recursive Procedures

Stefan Schwoon

Institut für Informatik
Technische Universität München

The Model:

- Pushdown Systems describe the control flow
- Binary Decision Diagrams represent variables

Definition of Pushdown Systems

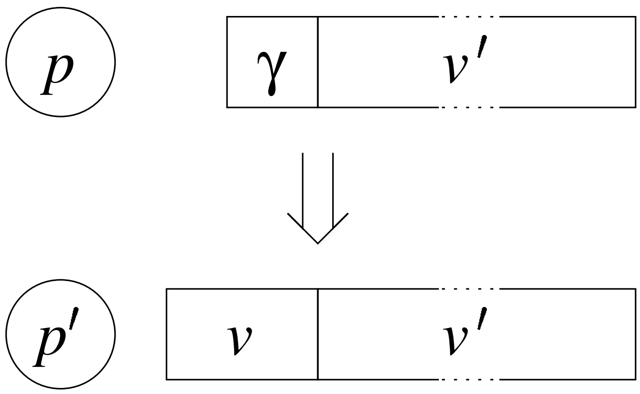
A pushdown system $\mathbf{P} = (P, \Gamma, c_0, \Delta)$ features

- a finite set of *control locations*: P
- a finite *stack alphabet*: Γ

  Call a pair $\langle p, v \rangle$, $p \in P$, $v \in \Gamma^*$
  a *configuration* of $\mathbf{P}$.

- an initial configuration: c_0
- a finite set of *rules*: $\Delta \quad (\subseteq (P \times \Gamma) \times (P \times \Gamma^*))$

$\langle p, \gamma \rangle \hookrightarrow \langle p', v \rangle \in \Delta$ means:



Normalisation: $0 \leq v \leq 2$

Pushdown Systems ...

... can model sequential programs (with finite data types) and recursive procedures.

Notation:

$\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle$ (simple statement)
$\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$ (procedure call)
$\langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle$ (return statement)

Interpretation:

$p, p', \ldots$ contain global control information and contents of *global* variables

$\gamma, \gamma', \gamma'', \ldots$ contain current location/return addresses and contents of *local* variables

An Example

A fragment of C:

```
int x;

void g(int a) {
  ...
}

void f() {
  int y;
  ...
  g(x + (y++));
}
```

The procedure call is modelled by a set of rules:

$$\langle\ (x),\ (f,y)\ \rangle \hookrightarrow \langle\ (x),\ (g,x+y)\ (f',y+1)\ \rangle$$

for all $x,y \in \mathrm{dom}(\texttt{int})$.

Known results

Complexity of model-checking (for fixed formula):

- Walukiewicz (1996):
  – modal μ-calculus is EXPTIME-complete
- Bouajjani, Esparza, Maler (1997):
  – LTL is polynomial
  – CTL is PSPACE-hard
- Esparza, Hansel, Rossmanith, Schwoon (2000):
  – LTL takes $O(|P|^2\,|\Delta|)$ time, $O(|P|\,|\Delta|)$ space

$\rightarrow$ LTL looks promising – but P and Δ can grow exponentially in the number of variables.

Let n = size of control flow,
c = cardinality of the data type
m_1 = number of global variables
m_2 = max number of local variables

The LTL checking procedure becomes
$O(n \cdot c^{3m_1+m_2})$ time, $O(n \cdot c^{m_1+m_2})$ space

Usage of BDDs

Restriction to Boolean Programs

- use boolean variables only ($c = 2$)
- represent integers by multiple boolean variables

Idea: Use BDDs to represent variable transitions.

$$\langle\ (x),\ (f,y)\ \rangle \hookrightarrow \langle\ (x),\ (g,x+y)\ (f',y+1)\ \rangle$$

becomes

$$\langle\cdot,f\rangle \hookrightarrow \langle\cdot,g\,f'\rangle$$
$$X' = X\ \wedge\ A' = X+Y\ \wedge\ Y'' = Y+1$$

Note: Local variables in different functions can be mapped to the same BDD variables.

Advantages:

- natural representation for control flow
  can be analysed with efficient algorithms
- (hopefully) compact representation of variables
  mapping exploits locality of variable scopes

Disadvantage:

- worst-case complexity becomes $O(|P|^2\,|\Delta| + |P|\,|\Delta|^2)$
  often offset by compact representation

Example 1

Parametrised program with n procedures
(taken from `bebop` checker by Ball/Rajamani)

```
bool g;

void main() {
  level_1();
  level_1();
  if (!g) {
    reach: skip;
  }
}
```

```
void level_i() {
  int (0..7) i;
  if (g) {
    i = 0;
    while (i < 7) i++;
  } else if (i < n) {
    level_{i+1}();
    level_{i+1}();
  }
  g = !g;
}
```

Question: Is the labelled statement reachable?

- No recursion – but finite-state approach would blow up
- exponential number of states – but linear solution
- $O(n)$ variables – but locality can be exploited

Example 1 – cont'd

The corresponding pushdown system:

```
global bool g;
local(l1a,l1b,l1c,l1d) int i(3);
local(l2a,l2b,l2c,l2d) int i(3);
...

(q <main0>)

q <main0> --> q <l1a main1>     (g' == g)
q <main1> --> q <l1a main2>     (g' == g)
q <main2> --> q <reach>         (!g)
q <main2> --> q <notreach>      (g)
q <reach> --> q <reach>
q <notreach> --> q <notreach>

q <l1a> --> q <l1b>       ((g' == g) & g & i' = 0)
q <l1b> --> q <l1b>       ((g' == g) & i < 7 & i' = i+1)
q <l1b> --> q <l1c>       ((g' == g) & i = 7 & i' = i)
q <l1a> --> q <l2a l1d>   ((g' == g) & !g & i = i'')
q <l1d> --> q <l2a l1c>   (g' == g & i = i'')
q <l1c> --> q <>          (g' == !g)

...
```

Example 1 – Results

Running time:

linear (in n) for the checking
for $n = 800$: $\sim 5.5\,\mathrm{s}$

Memory consumption:

linear (in n) for the PDS
peak number of live BDD nodes is
independent of n (~ 140)

Example 2: Quicksort

```
void qs (int left,int right)
{
  int lo,hi,piv;
  if (left >= right) return;
  piv = a[right]; lo = left; hi = right;

  while (lo <= hi)
  {
    if (a[hi] > piv) {
      hi--;
    } else {
      swap a[lo],a[hi];
      lo++;
    }
  }

  qs(left,hi);
  qs(lo,right);
}
```

Desired property: Termination

Example 2 – cont'd

```
--- START ---
q <main0>
q <qs0 (left=1 & lo=0 & hi=0 & right=2)
   main1>
    [ init ]
q <qs1 (left=1 & lo=1 & hi=2 & right=2)
   main1>
--- LOOP ---
    [ increase lo ]
q <qs1 (left=1 & lo=2 & hi=2 & right=2)
   main1>
    [ increase lo ]
q <qs1 (left=1 & lo=3 & hi=2 & right=2)
   main1>
    [ left recursive call ]
q <qs0 (left=1 & lo=0 & hi=0 & right=2)
   qs2 main1>
    [ init ]
q <qs1 (left=1 & lo=1 & hi=2 & right=2)
   qs2 main1>
```

Example 2 – Results

Using a corrected version of the algorithm:

bits	vars	time	nodes	states
2	10	0.14 s	8074	56
3	15	6.23 s	213580	792
4	20	–		

$\rightarrow$ Results spoilt by worse complexity of the algorithm

Idea: Limit the algorithm's state space by invariants

$$(\mathit{left} \leq \mathit{eq} \leq \mathit{lo},\ \mathit{hi} \leq \mathit{right})$$

New results:

bits	vars	time	nodes	states
2	10	0.11 s	1931	56
3	15	0.26 s	17467	792
4	20	2.73 s	159919	15504
5	25	62.61 s	427449	376692
6*	30	571.75 s	293946	10424128

Example 2 – PDS model

```
define N 3
local(qs0,qs1,qs2,qs3)
     int left(N), eq(N), lo(N), hi(N), right(N);

(q <main0>)

q <main0> --> q <qs0 main1>
q <main1> --> q <main1>

q <qs0> --> q <>        "reject"  (left >= right)

q <qs0> --> q <qs1>     "init"
       (left < right & left' = left & right' = right
        & eq' = left & lo' = left & hi' = right)

q <qs1> --> q <qs1>     "decrease hi"
       (lo <= hi & (hi != right | lo != left)
        & eq' = eq & lo' = lo & hi' = hi-1
        & left' = left & right' = right)

q <qs1> --> q <qs1>     "increase lo + eq"
       (lo <= hi & (hi != right | lo != left)
        & eq' = eq+1 & lo' = lo+1 & hi' = hi
        & left' = left & right' = right)

q <qs1> --> q <qs1>     "increase lo"
       (lo <= hi & eq' = eq & lo' = lo+1 & hi' = hi
        & left' = left & right' = right)

q <qs1> --> q <qs0 qs2> "left recursive call"
       (lo = hi+1 & left' = left & right' = eq-1
        & lo'' = lo & right'' = right)

q <qs2> --> q <qs0 qs3> "right recursive call"
       (left' = lo & right' = right)

q <qs3> --> q <>
```

Dataflow Analysis

- Used in optimizing compilers
- Algorithms for the intraprocedural case exist
- PDS provide a solution for interprocedural analysis

Example: Dead Variables

Def: A variable v is *dead* at program point n
if after leaving n it is either not used at all
or redefined before it is used again.

Usage: Elimination of useless variables

Solution using PDS and LTL:

Let Ref_v be true of program points
which use the value of variable v.

Let Def_v be true of program points
which assign a new value to v.

Variable v is *dead* at all states which satisfy

$$\mathbf{X}(\neg Ref_v \;\mathbf{U}\; (Def_v \vee Term))$$

Another approach: Solve problem using pre^* and $post^*$
(Esparza, Knoop)

Pointers

- Papers mentioned in the article
- SLAM/bebop project at Microsoft Research
- checker for modal μ-calculus by Daniel Polanský

Future work

- Optimisations
- Generalise the algorithms
- **more examples**