Infinite-State Backward Exploration of Boolean Broadcast Programs*

Peizun Liu and Thomas Wahl
Northeastern University, Boston, USA

FMCAD 2014
Lausanne, Switzerland

October 24, 2014

*This work is supported by NSF grant no. 1253331.

Outline

Problem Description

Assertion checking for non-recursive, unbounded-thread Boolean broadcast programs

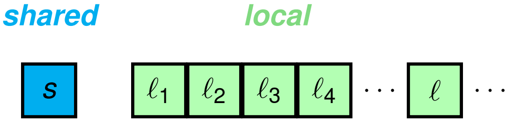
```
decl s := 0; // shared
main() {
      decl l := 0; // local
    1: s := 0;
    2: goto 3,7;
    3: assume(s);
    4: l := 1;
    5: wait;
    6: goto 7;
    7: assume(!s);
    8: broadcast;
    9: s := !s;
   10: assert(!l);
}
```

Problem Description

Definition

Given: *a program state* (s, ℓ)*, with shared component s and local component* ℓ

Task: *check if there exists a reachable global state of the form:*



Motivation

- Boolean broadcast programs result from concurrent C programs via predicate abstraction [Donaldson et al., 2012]
- Predicate abstraction used widely in verification: SLAM, BLAST, SATABS (concurrent), etc.

```
int x = 1;

int main() {
    int y = 0;

    x = 0;
    if(x)
      y = 1;
    x = !x;
    assert(!y);
    return 0;
}
```

```
decl s := 0;
main() {
      decl l := 0;
   1: s := 0;
   2: goto 3,6;
   3: assume(s);
   4: l := 1;
   5: goto 7;
   6: assume(!s);
   7: s := !s;
   8: assert(!l);
}
```

Motivation: Classical Solutions

Reachability of $(s, \ell) \Rightarrow$ *coverability problem*

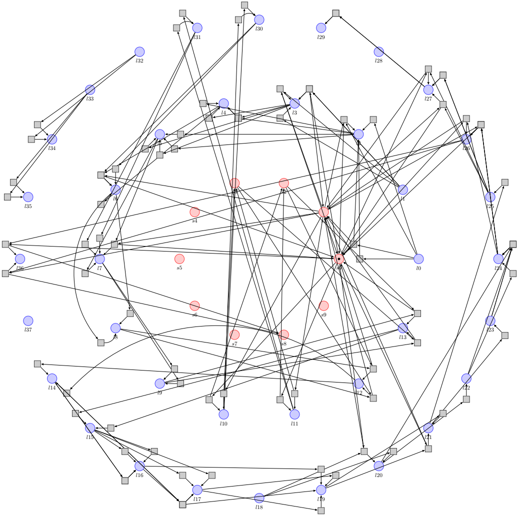
- Karp-Miller Procedure [Karp & Miller, 1969]
- Backward Search [Abdulla et al., 1996]

Limitations

- **Karp-Miller** procedure can not deal with broadcasts
- **Both** operate on transition systems
  $\Rightarrow$ need to first convert concurrent BP to *Petri net*

Motivation: State Space Blow-Up

Boolean Program to Petri Net: Program from Slide 5

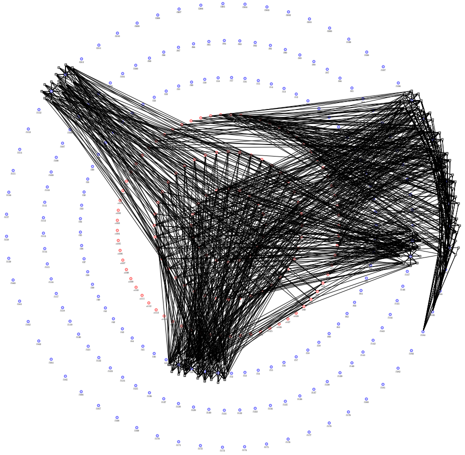


$|T| = 84$

Motivation: State Space Blow-Up

Boolean Program to Petri Net: one benchmark

BP: $|V_S| = 5$, $|V_L| = 2$, $LOC = 60$



$|T| = 8064$

Our Approach

Boolean broadcast program backward search

... based on Abdulla's Backward Search.

But:

- operates directly on Boolean program
- instead of statically building transition system, constructs it on-the-fly

Result: dramatic reduction of state explosion

Outline

Introduction

Classical BWS

Our Approach

Experiments

Summary

Backward Search [Abdulla et al., 1996]

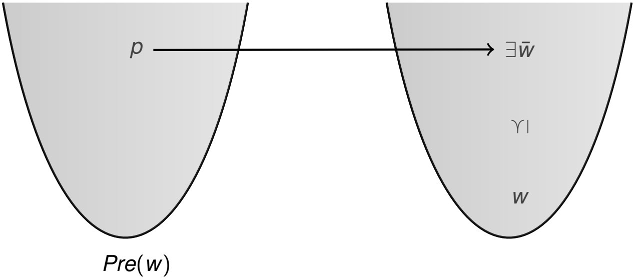
WQOS and cover relation

BWS operates over a *well quasi-ordered system* (WQOS).
In our case: WQO is the *covers* relation:

$$(s, \bar{\ell}_1, \ldots, \bar{\ell}_{\bar{n}}) \succeq (s, \ell_1, \ldots, \ell_n)$$

whenever $\mathit{multiset}\{\bar{\ell}_1, \ldots, \bar{\ell}_{\bar{n}}\} \supseteq \mathit{multiset}\{\ell_1, \ldots, \ell_n\}$.

Backward Search [Abdulla et al., 1996]



$$\mathrm{CovPre}(w) \quad = \quad \min \mathit{Pre}(w)$$

Outline

State Representation

Store states in *counter-abstracted form*:

$$\tau = \langle s, \{(\ell_1, n_1), \ldots, (\ell_k, n_k)\}\rangle$$

- $\ell_1, \ldots, \ell_k$ are the *distinct* local states occurring in τ
- n_i = # of threads in local state ℓ_i in τ ($n_i > 0$!)

Cover Predecessor Computation

$$\text{CovPre}(w) = \min\{p : \exists \bar{w} \succeq w : p \rightarrow \bar{w}\}$$

Two challenges:

1. given w, need to explore expanded elements $\bar{w} \succeq w$

   $\Rightarrow$ how many threads to be added?

2. given $\bar{w}$, need to compute predecessor: $p \rightarrow \bar{w}$
   We do not have $\rightarrow$, only the program $\mathcal{B}$!

   $\Rightarrow$ how to execute $\mathcal{B}$ backwards ?

Cover Predecessor Computation

Two challenges

1. need to expand w to $\bar{w}$
2. need to execute $\mathcal{B}$ backwards from $\bar{w}$

The solutions

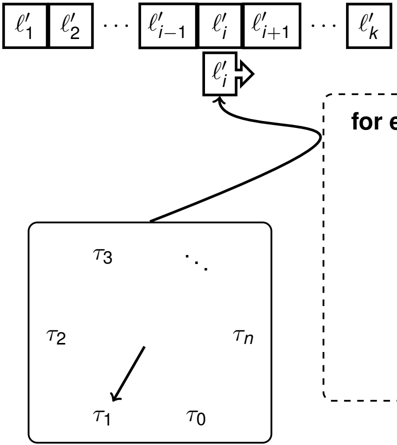
1. adding a **single** thread to w is sufficient[1]
2. execute $\mathcal{B}$ backwards via **WP** and **CFG**

[1] see paper for details

Our Algorithm: Standard Predecessors

$$\tau' = \langle s', \{(\ell'_1, n'_1), \ldots, (\ell'_k, n'_k)\}\rangle$$

Standard predecessors



= local states in τ'

for each CFG edge e s.t. $target(e) = \ell'_i.pc$

switch $e.stmt$**:**

case *sequential statement:*

...

case *thread creation statement:*

...

case *broadcast statement:*

...

Our Algorithm: Standard Predecessors

$$\tau' = \langle s', \{(\ell'_1, n'_1), \ldots, (\ell'_k, n'_k)\}\rangle$$

Sequential statements (e.g. assignments)

- compute the predecessors using $\text{WP}_{e.stmt}$:

  for each (s, ℓ) s.t. $\text{WP}_{e.stmt}(s, \ell, s', \ell'_i)$

  compute the predecessors of τ' w.r.t. (s, ℓ)

Our Algorithm: Standard Predecessors

$$\tau' = \langle s', \{\dots, (\ell'_i, n_i), \dots, (\ell'_j, n_j), \dots\}\rangle$$

Thread creation statement

```
   10: start_thread 20;
➡  11: ...
    ⋮
   20: ...
    ⋮
```

τ' has a predecessor iff there exists ℓ'_i, ℓ'_j in τ' s.t.

$$\begin{aligned} & \ell'_i.pc = 11 \\ \wedge \quad & \ell'_j.pc = 20 \\ \wedge \quad & \forall v \in V_L : \ell'_j.v = \ell'_i.v \end{aligned}$$

Predecessor:

$$\tau = \langle s', \{\dots, (\ell'_i, n_i - 1), \dots, (\ell'_j, n_j - 1), \dots, (\ell_k, n_k + 1), \dots\}\rangle$$

where $\ell_k.pc = 10 \wedge \forall v \in V_L : \ell_k.v = \ell'_i.v$

Our Algorithm: Standard Predecessors

$$\tau' = \langle s', \{\ldots, (\ell'_i, n_i), \ldots, (\ell'_j, n_j), \ldots, (\ell'_k, n_k)\}\rangle$$

Broadcast statement

First find

$$\ell'_i.pc = 31, \ell'_j.pc = 21, \ell'_k.pc = 11$$

```
    10: wait;
➡   11: ...
     ⋮
    20: wait;
➡   21: ...
     ⋮
    30: broadcast;
➡   31: ...
     ⋮
```

Our Algorithm: Standard Predecessors

Broadcast statement

Current State

```
    10: wait;
➦   11: ...
     ⋮
    20: wait;
➦   21: ...
     ⋮
    30: broadcast;
➠   31: ...
     ⋮
```

broadcast ←

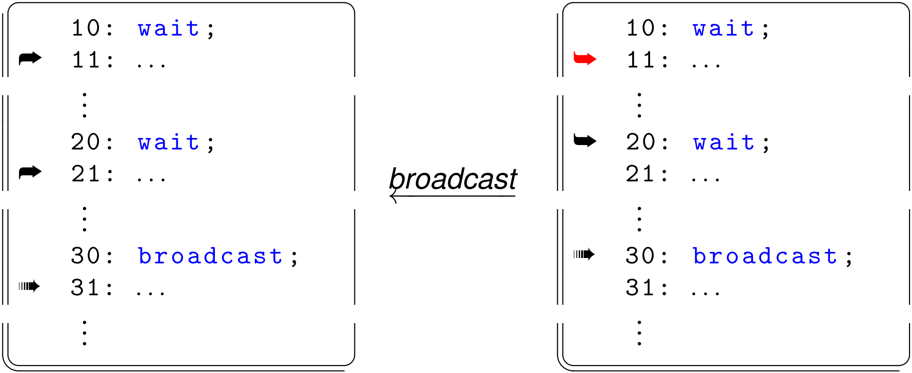
Predecessor could be ...

```
➥   10: wait;
    11: ...
     ⋮
➥   20: wait;
    21: ...
     ⋮
➠   30: broadcast;
    31: ...
     ⋮
```

Our Algorithm: Standard Predecessors

Broadcast statement

Current State *Predecessor could be ...*



Our Algorithm: Standard Predecessors

Broadcast statement

Current State

```
   10: wait;
➡  11: ...
    ⋮
   20: wait;
➡  21: ...
    ⋮
   30: broadcast;
➡  31: ...
    ⋮
```

broadcast ←

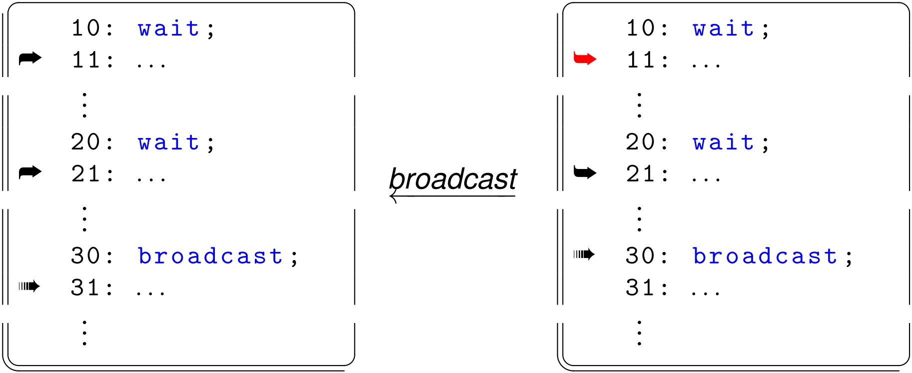
Predecessor could be ...

```
➡  10: wait;
   11: ...
    ⋮
   20: wait;
➡  21: ...
    ⋮
➡  30: broadcast;
   31: ...
    ⋮
```

Our Algorithm: Standard Predecessors

Broadcast statement

Current State *Predecessor could be ...*



Our Algorithm: Standard Predecessors

$$\tau' = \langle s', \{\ldots, (\ell'_i, n_i), \ldots, (\ell'_j, n_j), \ldots, (\ell'_k, n_k)\}\rangle$$

Broadcast statement

First find

$$\ell'_i.pc = 31, \ell'_j.pc = 21, \ell'_k.pc = 11$$

```
   10: wait;
➡  11: ...
    ⋮
   20: wait;
➡  21: ...
    ⋮
   30: broadcast;
➡  31: ...
    ⋮
```

Predecessors: Each subset of past-wait threads gives rise to a different predecessor

Our Algorithm: Expanded Predecessors

$$\tau' = \langle s', \{(\ell'_1, n'_1), \ldots, (\ell'_k, n'_k)\}\rangle$$

Expanded predecessors

for each (s, ℓ) s.t. $\exists m' \notin \{\ell'_1, \ldots, \ell'_k\}$:

$e := (\ell.pc, m'.pc) \in CFG$

$\wedge$ *e.stmt* may modify the shared state

$\wedge\ \mathrm{WP}_{e.stmt}(s, \ell, s', m')$

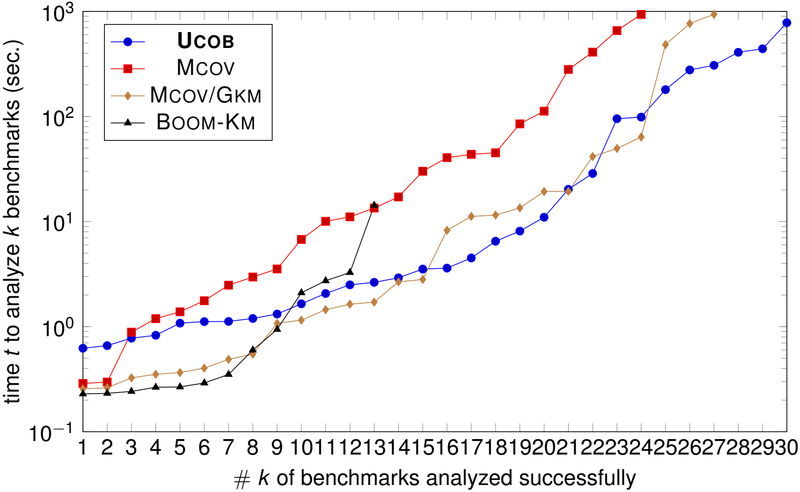
compute the predecessors of τ' w.r.t. (s, ℓ)

Outline

Experiments: Benchmark Sample

ID/Program	C Program				Boolean Program				Safe?
	SV	LV	LOC	Bc?	$\|V_S\|$	$\|V_L\|$	Its.	Mod.Sh.	
01/INC-L	2	1	46	○	3	1	2	7.5	●
02/INC-C	1	3	57	○	0	4	4	0	●
03/PRNSIMP-L	2	4	63	○	2	3	2	7.7	●
04/PRNSIMP-C	1	5	95	○	0	5	2	0	●
05/BS-LOOP	0	6	24	○	0	7	1	0	○
06/PTHREAD	5	0	85	○	7	0	5	17.1	○
07/MAXOPT-L	3	4	69	○	1	1	2	3.1	●
08/MAXOPT-C	2	6	86	○	0	2	2	0	●
09/STACK-L	4	2	79	○	1	3	3	3.8	●
10/STACK-C	3	3	89	○	3	1	2	6.4	●
11/BSD-AK	1	7	90	●	3	1	15	11.7	●
12/BSD-RA	2	21	87	●	3	0	19	12.3	●
13/NETBSD	1	28	152	●	3	1	30	10.1	●
14/SOLARIS	1	56	122	●	5	1	14	10.9	●
15/BOOP	5	2	89	○	5	2	4	11.4	○
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Experimental Results



Outline

Summary

Our approach

- avoids the static transition system construction
- operates on-the-fly: what you see is what you pay
- can result in dramatic savings

Thank You

References

- A. Donaldson, A. Kaiser, D. Kroening, M. Tautschnig, and T. Wahl, “Counterexample-guided abstraction refinement for symmetric concurrent programs,” *Form. Method. Syst. Des.*, 2012.
- R. M. Karp and R. E. Miller, “Parallel program schemata,” *J. Comput. Syst. Sci.*, 1969.
- P. Abdulla, K. Cerans, B. Jonsson, and Y. Tsay, “General decidability theorems for infinite-state systems,” in *LICS*, 1996.
- A. Kaiser, D. Kroening, and T. Wahl, “Efficient coverability analysis by proof minimization,” in *CONCUR*, 2012.