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

Introduction

Classical BWS

Our Approach

Experiments

Summary
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 \((s, \ell)\), with shared component \(s\) and local component \(\ell\)

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

![Diagram of shared and local components](image-url)
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.

```c
int x = 1;
int main () {
    int y = 0;
    x = 0;
    if(x)
        y = 1;
    x = !x;
    assert (!y);
    return 0;
}
```

```c
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 \((s, \ell)\) ⇒ *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
  ⇒ 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:

\[(s, \overline{\ell}_1, \ldots, \overline{\ell}_n) \succeq (s, \ell_1, \ldots, \ell_n)\]

whenever \(\text{multiset}\{\overline{\ell}_1, \ldots, \overline{\ell}_n\} \supseteq \text{multiset}\{\ell_1, \ldots, \ell_n\}\).
Backward Search [Abdulla et al., 1996]

\[ \exists \bar{w} \preceq w \text{ CovPre}(w) = \min \text{ Pre}(w) \]
Outline

Introduction

Classical BWS

Our Approach

Experiments

Summary
State Representation

Store states in \textit{counter-abstracted form}:

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

- \( \ell_1, \ldots, \ell_k \) are the \textit{distinct} local states occurring in \( \tau \)
- \( n_i = \# \) of threads in local state \( \ell_i \) in \( \tau \) (\( n_i > 0 \) !)
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 $B$ !
   $\Rightarrow$ how to execute $B$ backwards ?
Two challenges

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

The solutions

1. adding a **single** thread to $w$ is sufficient\(^1\)
2. execute $B$ 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

\[
\ell'_1 \quad \ell'_2 \quad \ldots \quad \ell'_{i-1} \quad \ell'_i \quad \ell'_{i+1} \quad \ldots \quad \ell'_k
\]

= local states in \( \tau' \)

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

\begin{align*}
\text{switch} & \ e.\text{stmt}: \\
\text{case} & \ \text{sequential statement}: \\
& \quad \ldots \\
\text{case} & \ \text{thread creation statement}: \\
& \quad \ldots \\
\text{case} & \ \text{broadcast statement}: \\
& \quad \ldots
\end{align*}
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 \( WP_{e.stmt} \):

  for each \((s, \ell)\) s.t. \( WP_{e.stmt}(s, \ell, s', \ell'_i) \)
  
  compute the predecessors of \( \tau' \) w.r.t. \((s, \ell)\)
Our Algorithm: Standard Predecessors

\[ \tau' = \langle s', \{ \ldots, (\ell'_i, n_i), \ldots, (\ell'_j, n_j), \ldots \} \rangle \]

Thread creation statement

```
10: start_thread 20;
11: ...
\vdots
20: ...
```

\( \tau' \) has a predecessor iff there exists \( \ell'_i, \ell'_j \) in \( \tau' \) s.t.

\[
\ell'_i.pc = 11 \land \ell'_j.pc = 20 \land \forall v \in V_L: \ell'_j.v = \ell'_i.v
\]

Predecessor:

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

where \( \ell_k.pc = 10 \land \forall v \in V_L: \ell_k.v = \ell'_i.v \)
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

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

\[
\begin{align*}
10: & \quad \text{wait;}
11: & \quad \ldots \\
\vdots
20: & \quad \text{wait;}
21: & \quad \ldots
\vdots
30: & \quad \text{broadcast;}
31: & \quad \ldots
\vdots
\end{align*}
\]
Our Algorithm: Standard Predecessors

Broadcast statement

Current State

10: wait;
11: ...
...
20: wait;
21: ...
...
30: broadcast;
31: ...
...

Predecessor could be ...

10: wait;
11: ...
...
20: wait;
21: ...
...
30: broadcast;
31: ...
...

broadcast
**Our Algorithm: Standard Predecessors**

**Broadcast statement**

<table>
<thead>
<tr>
<th>Current State</th>
</tr>
</thead>
<tbody>
<tr>
<td>10: wait;</td>
</tr>
<tr>
<td>11: ...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>20: wait;</td>
</tr>
<tr>
<td>21: ...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>30: broadcast;</td>
</tr>
<tr>
<td>31: ...</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

**Predecessor could be ...**

| 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: ...
...

Predecessor could be ...

10: wait;
11: ...
...
20: wait;
21: ...
...
30: broadcast;
31: ...
...
Our Algorithm: Standard Predecessors

Broadcast statement

Current State

10: \texttt{wait};
11: ... 
... 
20: \texttt{wait};
21: ... 
... 
30: \texttt{broadcast};
31: ... 
... 

Predecessor could be ...

10: \texttt{wait};
11: ... 
... 
20: \texttt{wait};
21: ... 
... 
30: \texttt{broadcast};
31: ... 
...
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;  
<table>
<thead>
<tr>
<th>\arrowup</th>
<th>11: ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>\vdots</td>
<td>\vdots</td>
</tr>
</tbody>
</table>
| 20: wait;  
|\arrowup| 21: ... |
|\vdots|\vdots|
| 30: broadcast;  
|\arrowup| 31: ... |
|\vdots|\vdots|

**Predecessors:** Each subset of past-wait threads gives rise to a different predecessor
Our Algorithm: Expanded Predecessors

\[ \tau' = \langle s', \{(l_1', n_1'), \ldots, (l_k', n_k')\} \rangle \]

Expanded predecessors

for each \((s, \ell)\) s.t. \(\exists m' \notin \{l_1', \ldots, l_k'\}\) :
\[ e := (\ell.pc, m'.pc) \in CFG \]
\[ \land e\text{-stmt} \text{ may modify the shared state} \]
\[ \land WP_{e\text{-stmt}}(s, \ell, s', m') \]
compute the predecessors of \(\tau'\) w.r.t. \((s, \ell)\)
## Experiments: Benchmark Sample

<table>
<thead>
<tr>
<th>ID/Program</th>
<th>C Program</th>
<th>Boolean Program</th>
<th>Safe?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SV</td>
<td>LV</td>
<td>LOC</td>
</tr>
<tr>
<td>01/INC-L</td>
<td>2</td>
<td>1</td>
<td>46</td>
</tr>
<tr>
<td>02/INC-C</td>
<td>1</td>
<td>3</td>
<td>57</td>
</tr>
<tr>
<td>03/PRNSIMP-L</td>
<td>2</td>
<td>4</td>
<td>63</td>
</tr>
<tr>
<td>04/PRNSIMP-C</td>
<td>1</td>
<td>5</td>
<td>95</td>
</tr>
<tr>
<td>05/BS-LOOP</td>
<td>0</td>
<td>6</td>
<td>24</td>
</tr>
<tr>
<td>06/PTHREAD</td>
<td>5</td>
<td>0</td>
<td>85</td>
</tr>
<tr>
<td>07/MAXOPT-L</td>
<td>3</td>
<td>4</td>
<td>69</td>
</tr>
<tr>
<td>08/MAXOPT-C</td>
<td>2</td>
<td>6</td>
<td>86</td>
</tr>
<tr>
<td>09/STACK-L</td>
<td>4</td>
<td>2</td>
<td>79</td>
</tr>
<tr>
<td>10/STACK-C</td>
<td>3</td>
<td>3</td>
<td>89</td>
</tr>
<tr>
<td>11/BSD-AK</td>
<td>1</td>
<td>7</td>
<td>90</td>
</tr>
<tr>
<td>12/BSD-RA</td>
<td>2</td>
<td>21</td>
<td>87</td>
</tr>
<tr>
<td>13/NETBSD</td>
<td>1</td>
<td>28</td>
<td>152</td>
</tr>
<tr>
<td>14/SOLARIS</td>
<td>1</td>
<td>56</td>
<td>122</td>
</tr>
<tr>
<td>15/BOOP</td>
<td>5</td>
<td>2</td>
<td>89</td>
</tr>
</tbody>
</table>
Experimental Results

![Graph showing experimental results](image_url)

- $\text{UCOB}$
- $\text{MCOV}$
- $\text{MCOV/GKM}$
- $\text{BOOM-KM}$

The graph plots the time $t$ to analyze $k$ benchmarks against the number of benchmarks $k$ analyzed successfully. The x-axis represents the number of benchmarks analyzed, while the y-axis represents the time in seconds on a logarithmic scale.
Outline

Introduction

Classical BWS

Our Approach

Experiments

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


