Response property checking via distributed state space exploration

Brad Bingham and Mark Greenstreet
{binghamb, mrg}@cs.ubc.ca

Department of Computer Science
University of British Columbia, Canada

October 24, 2014

FMCAD 2014
Motivation: Liveness + Explicit-State

- High-Level Models: use Mur$\varphi$ to describe a system
- Liveness: nice to verify, but challenging in practice
- Distributed Model Checking: memory and speed scalability
- Explicit-State: easy to distribute/parallelize
  - (Also outperforms symbolic methods for certain models)

**Our Goal:** Attack a practical liveness property called response with distributed, explicit-state model checking
Outline

1. Response and Fairness

2. High Level Algorithm

3. Our Implementation
   - Distributed MC for Safety
   - Adaptation for Response
   - One Optimization (of many)

4. Results
“Will there always be a response?” $\equiv$ “Does every fair path from each reachable $p$-state lead to a $q$-state?”

- $p \equiv$ “request issued”; $q \equiv$ “request granted”
- In LTL: $fair \Rightarrow \Box(p \rightarrow \Diamond q)$
- Most common/simplest notion of liveness
“Will there always be a response?” ≡ “Does every fair path from each reachable $p$-state lead to a $q$-state?”

- $p$ ≡ “request issued”; $q$ ≡ “request granted”
- In LTL: $\text{fair} \Rightarrow \square(p \rightarrow \Diamond q)$
- Most common/simplest notion of liveness
**pending** ≡ “states where the request is outstanding”

The question $\text{fair} \Rightarrow \Box(p \rightarrow \Diamond q)$? Is equivalent to asking “Is there a fair SCC within $\text{pending}$?”

- Terminology: fair SCC $\equiv$ FSCC
In practice, we use fairness assumptions that reflect the underlying implementation.

- Excludes unrealistic counterexamples
- We use action-based fairness:
  - An action $a$ is a set of system transitions
  - $a$ is called strongly-fair (aka compassionate; $a \in \mathcal{C}$) if
    \[ [a \text{ enabled } \infty\text{-often}] \Rightarrow [a \text{ fires } \infty\text{-often}] \]
  - $a$ is called weakly-fair (aka just; $a \in \mathcal{J}$) if
    \[ [a \text{ presientently enabled}] \Rightarrow [a \text{ fires}] \]
In practice, we use fairness assumptions that reflect the underlying implementation.

Excludes unrealistic counterexamples

We use action-based fairness:

- An action $a$ is a set of system transitions
- $a$ is called strongly-fair (aka compassionate; $a \in C$) if
  $[ a \text{ enabled } \infty\text{-often} ] \Rightarrow [ a \text{ fires } \infty\text{-often} ]$
- $a$ is called weakly-fair (aka just; $a \in J$) if
  $[ a \text{ presistently enabled} ] \Rightarrow [ a \text{ fires} ]$

Note: verifying $fair \Rightarrow \Box(p \rightarrow \Diamond q)$ with standard Büchi automata

LTL MC approach will blow up

- *i.e.*, property automata with size exponential in $|C \cup J|$
1. Response and Fairness

2. High Level Algorithm

3. Our Implementation
   - Distributed MC for Safety
   - Adaptation for Response
   - One Optimization (of many)

4. Results
Both green actions and pink actions are strongly fair
Both green actions and pink actions are strongly fair

FSCCs
Both green actions and pink actions are strongly fair
Both *green* actions and *pink* actions are strongly fair

Purple Blob $\equiv$ MaybeFair

Pending states: $a$, $b$, $c$, $d$, $e$, $f$, $g$, $h$
Both **green** actions and **pink** actions are strongly fair

Purple Blob $\equiv$ MaybeFair

**Idea:** find unfair states by looking at previous actions within $\langle$MaybeFair$\rangle$
Both **green** actions and **pink** actions are **strongly fair**

Purple Blob $\equiv$ MaybeFair

Idea: find unfair states by looking at previous actions within $\langle$MaybeFair$\rangle$
Algorithm Example

Both green actions and pink actions are strongly fair

Purple Blob ≡ MaybeFair

Idea: find unfair states by looking at previous actions within \langle MaybeFair \rangle
Both *green* actions and *pink* actions are *strongly fair*

Purple Blob ≡ MaybeFair

**Idea**: find unfair states by looking at previous actions within $\langle \text{MaybeFair} \rangle$
Both **green** actions and **pink** actions are **strongly fair**

Purple Blob $\equiv$ MaybeFair

Idea: find unfair states by looking at previous actions within $\langle$MaybeFair$\rangle$
Both green actions and pink actions are strongly fair.

Purple Blob \equiv MaybeFair

Idea: find unfair states by looking at previous actions within \langle MaybeFair \rangle
Definition: Predecessor Actions (PAs)

- Suppose $H \subseteq \text{pending}$. Let $\langle H \rangle$ be the subgraph of the transition graph induced by $H$.
- The **Predecessor Actions** for state $s \in H$, are actions appearing on some path that
  1. is contained within $\langle H \rangle$; and
  2. ends at $s$.
- **Observe**: If $s$ lies on a FSCC in $\langle H \rangle$, then all enabled strongly-fair actions at $s$ are PAs.
- **Contrapositive**: If there $\exists$ a strongly-fair action enabled at $s$ that isn’t a PA, then $s$ does NOT lie on a FSCC in $\langle H \rangle$. 
Definition: Predecessor Actions (PAs)

- Suppose $H \subseteq \text{pending}$. Let $\langle H \rangle$ be the subgraph of the transition graph induced by $H$.
- The **Predecessor Actions** for state $s \in H$, are actions appearing on some path that
  1. is contained within $\langle H \rangle$; and
  2. ends at $s$.
- **Observe**: If $s$ lies on a FSCC in $\langle H \rangle$, then all enabled strongly-fair actions at $s$ are PAs.
- **Contrapositive**: If there $\exists$ a strongly-fair action enabled at $s$ that isn’t a PA, then $s$ does **NOT** lie on a FSCC in $\langle H \rangle$.

...and $\therefore$ remove $s$ from consideration!
Outline

1. Response and Fairness

2. High Level Algorithm

3. Our Implementation
   - Distributed MC for Safety
   - Adaptation for Response
   - One Optimization (of many)

4. Results
Distributed MC[SD97] Overview

- Simple approach to distributing explicit-state model checking (for safety)
  - Use uniform random hash function \( \text{owner} : \text{States} \rightarrow \text{PIDs} \)
  - PID \( i \) only stores states \( s \) such that \( \text{owner}(s) = i \).
- Each PID maintains two data structures:
  - \( V \): Set of (owned) states visited so far
  - \( WQ \): List of states waiting to be expanded
- Start: compute initial states and send to their owners
- Iterate: state successors are sent to their respective owners
- Termination: when each \( WQ \) is empty and no messages are in flight
Message Flow

WARE PROCESS $i$

$V$: \{ $s_1, \ldots, s_k$ \}

(visited states)

state $s$

where $\text{owner}(s) = i$

LAN/NoC to other Processes
Message Flow

**WORKER PROCESS** \( i \)

\[ V: \{s_1, \ldots, s_k\} \cup \{s\} \]

(visited states)

- if \( s \in V \rightarrow \) discard \( s \)
- if \( s \notin V \rightarrow \) add \( s \) to \( V \)

LAN/NoC to other Processes

state \( s \)

where \( \text{owner}(s) = i \)
Message Flow

WORKER PROCESS \( i \)

\[ V: \{s_1, \ldots, s_k\} \cup \{s\} \]

(visited states)

\[
\begin{align*}
\text{if } s \in V & \rightarrow \text{discard } s \\
\text{if } s \notin V & \rightarrow \text{add } s \text{ to } V
\end{align*}
\]

compute successors of \( s \)

\[ s'_1, \ldots, s'_r \]

state \( s \)

where \( \text{owner}(s) = i \)

\[ \{s'_1, \ldots, s'_r\} \]

\[ \text{owner}(s'_1), \ldots, \text{owner}(s'_r) \]
Hash Table Considerations

- **For safety:** use a Murφ hash table implementation that stores visited states as 40-bit values
  - Chance of a missed state, but typically it’s a tiny chance ($\approx 10^{-10}$)
  - Once a state is inserted, it can’t be recovered from its hash value
- **For response:** necessary to track extra information about states, for example
  - Is it a *pending*-state?
  - Is it in *MaybeFair*?
  - What are its predecessor actions, relative to $\langle \text{MaybeFair} \rangle$?
- We use $\approx 16 + |C \cup J|$ extra bits per state
Suppose $\mathcal{C} = \{a_1, \ldots, a_k\}$

“Tag” each hash table entry with PAs, which is a subset of $\mathcal{C}$

(plus a few other bookkeeping bits)

For states in $s \in \text{MaybeFair}$: initialize $PA(s)$ to $\emptyset$

Message Passing:

- Expand state $s$: if $(s, s') \in a_i$, send msg $[s', PA(s) \cup \{a_i\}]$ to $\text{owner}(s')$
- Receive msg $[s', F]$: $PA(s') := PA(s') \cup F$; expand state $s'$ if $PA(s')$ changed.
- Continue until no further expansions.
Suppose $\mathcal{C} = \{a_1, \ldots, a_k\}$

"Tag" each hash table entry with PAs, which is a subset of $\mathcal{C}$

(plus a few other bookkeeping bits)

For states in $s \in \text{MaybeFair}$: initialize $\mathit{PA}(s)$ to $\emptyset$

Message Passing:

- Expand state $s$: if $(s, s') \in a_i$, send msg $[s', \mathit{PA}(s) \cup \{a_i\}]$ to $\text{owner}(s')$
- Receive msg $[s', F]$: $\mathit{PA}(s') := \mathit{PA}(s') \cup F$; expand state $s'$ if $\mathit{PA}(s')$ changed.
- Continue until no further expansions.

(A similar idea works for weakly-fair actions)
Strongly-fair actions $C = \{a_1, \ldots, a_7\}$
PA Propagation Example

- Strongly-fair actions $\mathcal{C} = \{a_1, ..., a_7\}$
Strongly-fair actions $\mathcal{C} = \{a_1, \ldots, a_7\}$
PA Propagation Example

- Strongly-fair actions \( \mathcal{C} = \{ a_1, \ldots, a_7 \} \)
Strongly-fair actions \( C = \{a_1, \ldots, a_7\}\)
PA Propagation Example

- **Strongly-fair actions** $\mathcal{C} = \{a_1, \ldots, a_7\}$
• Strongly-fair actions $\mathcal{C} = \{a_1, \ldots, a_7\}$
Strongly-fair actions $C = \{a_1, \ldots, a_7\}$

(once PAs reach a fixpoint, remove unfair states from $MaybeFair$, clear the PAs and compute them again)
Optimization: The “Kernel”

**Idea:** save set of states $K$ to disk so that $MaybeFair$ can be generated through reachability starting with $K$

- Call $K$ a kernel if $MaybeFair \subseteq Reach(K)$
  - i.e., $MaybeFair$ is reachable starting from $K$
- Note: both initial states $I$ and $p$-states are kernels for all subsets of $pending$
- To maintain $K$:
  - Initialize $K$ to $p$-states;
  - If $s \in K$ is removed from $MaybeFair$, then
    - Remove $s$ from $K$;
    - Insert $successors(s) \cap MaybeFair$ into $K$
Kernel Optimization

MaybeFair

kernel

Response property checking
October 24/2014
Kernel Optimization

MaybeFair

pending

kernel
Kernel Optimization

pending kernel

MaybeFair

response property checking

October 24/2014
Kernel Optimization

pending

kernel

MaybeFair
Kernel Optimization

Pending
1 Response and Fairness

2 High Level Algorithm

3 Our Implementation
   - Distributed MC for Safety
   - Adaptation for Response
   - One Optimization (of many)

4 Results
## Performance

<table>
<thead>
<tr>
<th>model</th>
<th>runtime*</th>
<th>states†</th>
<th>(\mid\text{pending}\mid)†</th>
<th>exp/state</th>
</tr>
</thead>
<tbody>
<tr>
<td>german5_sf</td>
<td>189</td>
<td>15.8</td>
<td>4.9</td>
<td>3.48</td>
</tr>
<tr>
<td>german6_sf</td>
<td>4253</td>
<td>316.5</td>
<td>95.3</td>
<td>3.33</td>
</tr>
<tr>
<td>peterson6_wf</td>
<td>820</td>
<td>13.8</td>
<td>12.1</td>
<td>12.91</td>
</tr>
<tr>
<td>peterson7_wf</td>
<td>26957</td>
<td>380.3</td>
<td>340.5</td>
<td>14.19</td>
</tr>
<tr>
<td>snoop2_sf</td>
<td>160</td>
<td>2.6</td>
<td>1.3</td>
<td>12.71</td>
</tr>
<tr>
<td>saw20_sf</td>
<td>323</td>
<td>0.3</td>
<td>0.3</td>
<td>44.06</td>
</tr>
<tr>
<td>gbn3_2_sf</td>
<td>369</td>
<td>12.8</td>
<td>7.9</td>
<td>6.44</td>
</tr>
<tr>
<td>swp4_2_sf</td>
<td>503</td>
<td>18.6</td>
<td>11.7</td>
<td>6.58</td>
</tr>
<tr>
<td>intelsmall_sf</td>
<td>285</td>
<td>0.5</td>
<td>0.3</td>
<td>6.36</td>
</tr>
<tr>
<td>intelmed_sf</td>
<td>1,015</td>
<td>2.7</td>
<td>1.9</td>
<td>8.59</td>
</tr>
<tr>
<td>intelbig_sf</td>
<td>13,872</td>
<td>51.8</td>
<td>29.9</td>
<td>11.92</td>
</tr>
</tbody>
</table>

* runtime is in seconds; † state counts in millions

- **Blue**: 40 processes running on 20 Core i7 machines (UBC)
- **Green**: 16 processes running on Xeon machines (Intel)
Our Goal: Attack a practical liveness property called response with distributed, explicit-state model checking

Result: An efficient implementation for response property verification, applicable to very large state spaces
Our Goal: Attack a practical liveness property called \texttt{response} with distributed, explicit-state model checking

Result: An efficient implementation for response property verification, applicable to very large state spaces

- Our approach does well in practice – expands each state a small number of times (modest overhead compared with safety 😊)
  - (in the worst case, could expand each state $O(mn^2)$ times where $m$ is \# of fair rules and $n$ number of states)

- Optimizations improve the performance by more than a factor of 2 on average

- Our tool is massively scalable – can use on industrial problems
**Take-Away**

**Our Goal:** Attack a practical liveness property called **response** with distributed, explicit-state model checking

**Result:** An efficient implementation for response property verification, applicable to very large state spaces

- Our approach does well in practice – expands each state a small number of times (modest overhead compared with safety 😊)
  - (in the worst case, could expand each state $O(mn^2)$ times where $m$ is # of fair rules and $n$ number of states)

- Optimizations improve the performance by more than a factor of 2 on average

- Our tool is massively scalable – can use on industrial problems

**Thank-you!**
Our Goal: Attack a practical liveness property called response with distributed, explicit-state model checking

Result: An efficient implementation for response property verification, applicable to very large state spaces

- Our approach does well in practice – expands each state a small number of times (modest overhead compared with safety 😊)
  - (in the worst case, could expand each state $O(mn^2)$ times where $m$ is # of fair rules and $n$ number of states)

- Optimizations improve the performance by more than a factor of 2 on average

- Our tool is massively scalable – can use on industrial problems

Thank-you! Questions?