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

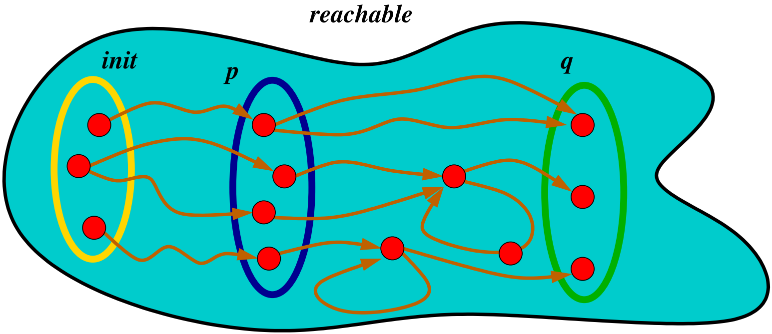
Motivation: Liveness + Explicit-State

- High-Level Models: use Murφ 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

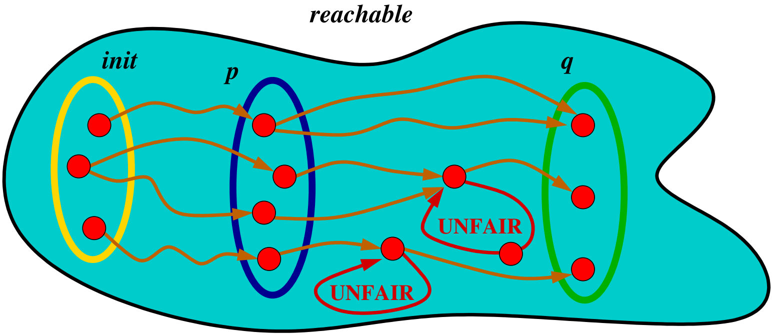
Outline

Response Properties



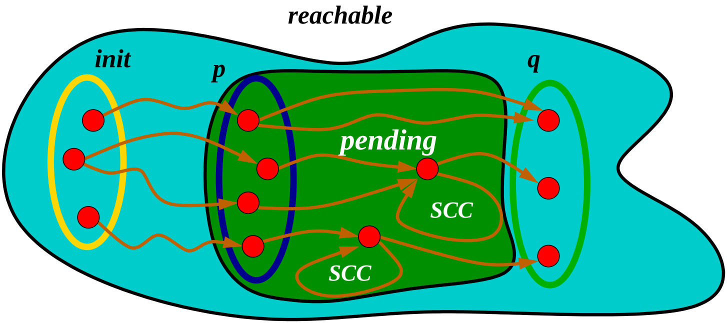
- "Will there always be a response?" $\equiv$ "Does every **fair** path from each reachable p-state lead to a q-state?"
  - $p \equiv$ "request issued"; $q \equiv$ "request granted"
  - In LTL: $fair \Rightarrow \Box(p \rightarrow \Diamond q)$
  - Most common/simplest notion of liveness

Response Properties



- “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

Response and Strongly Connected Components (SCCs)



- *pending* $\equiv$ "states where the request is outstanding"
- The question $fair \Rightarrow \Box(p \rightarrow \Diamond q)$? Is equivalent to asking "Is there a fair SCC within *pending*?"
  - Terminology: fair SCC $\equiv$ FSCC

Fairness

- 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* enabled ∞-often] $\Rightarrow$ [*a* fires ∞-often]
  - *a* is called weakly-fair (aka just; $a \in \mathcal{J}$) if
    [*a* presistently enabled] $\Rightarrow$ [*a* fires]

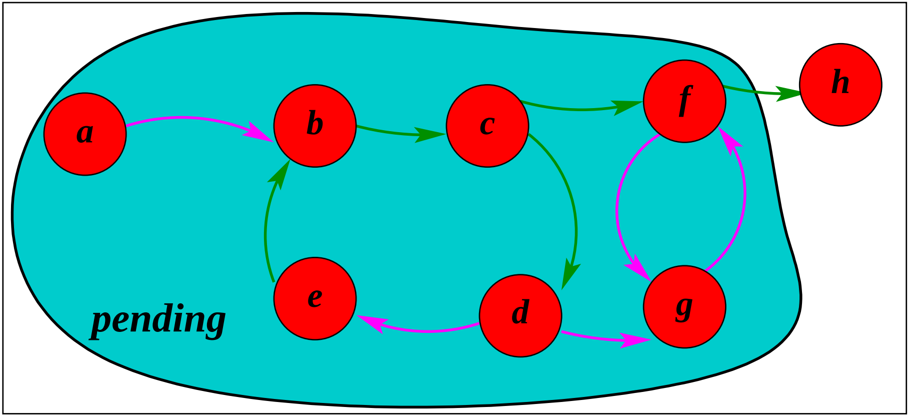
Fairness

- 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 enabled ∞-often] $\Rightarrow$ [a fires ∞-often]
  - a is called weakly-fair (aka just; $a \in \mathcal{J}$) if
    [a presistently enabled] $\Rightarrow$ [a fires]
- Note: verifying $\mathit{fair} \Rightarrow \Box(p \rightarrow \Diamond q)$ with standard Büchi automata LTL MC approach will blow up
  - *i.e.*, property automata with size exponential in $|\mathcal{C} \cup \mathcal{J}|$

Outline

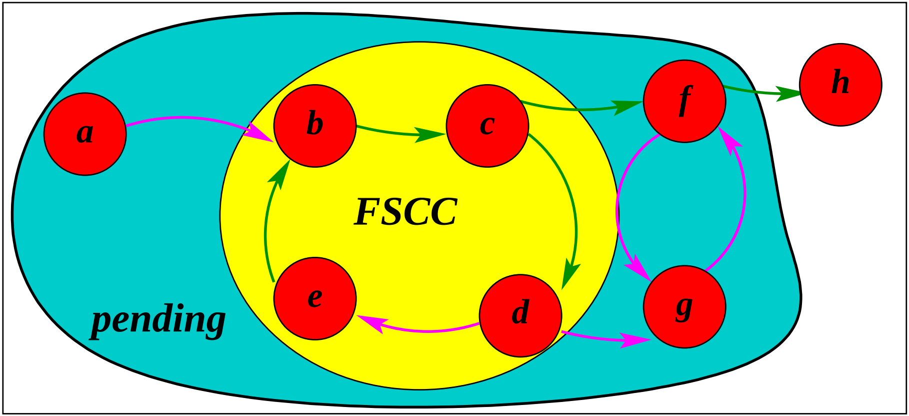
FSCCs

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



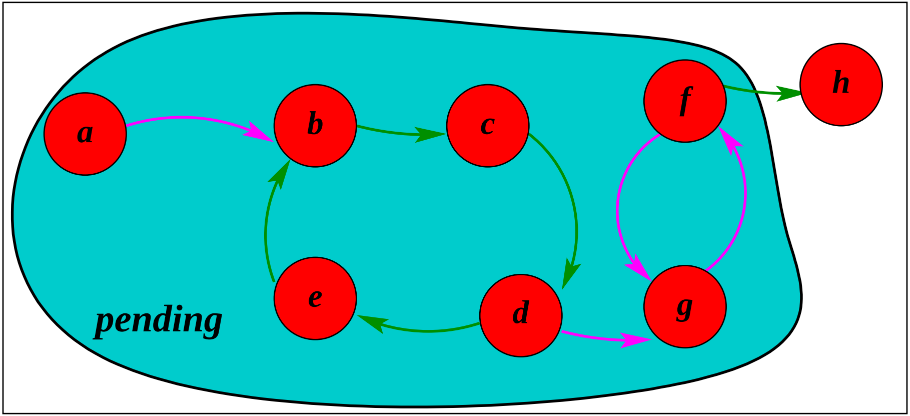
FSCCs

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



Algorithm Example

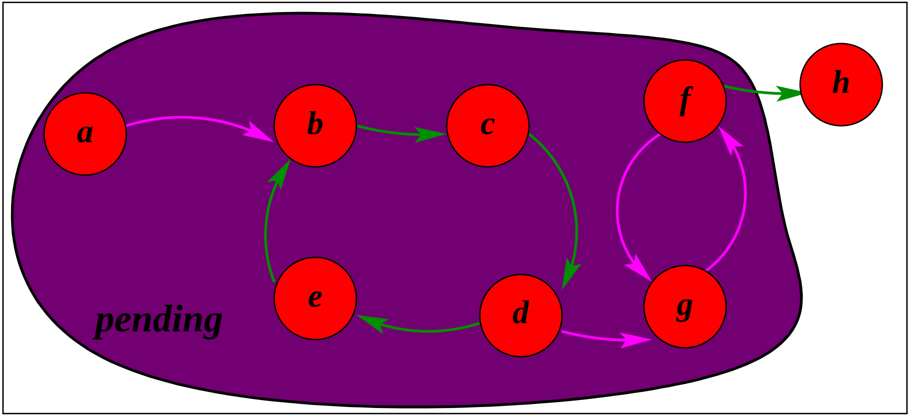
Both **green** actions and **pink** actions are **strongly fair**



Algorithm Example

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

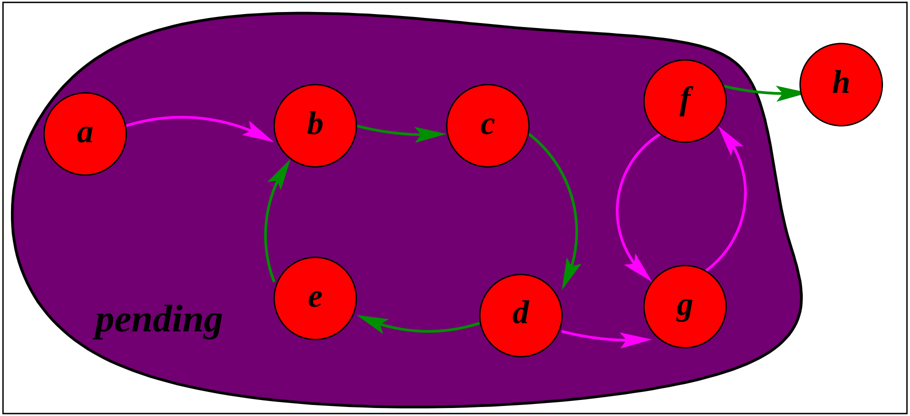
Purple Blob $\equiv$ *MaybeFair*



Algorithm Example

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

Purple Blob $\equiv$ *MaybeFair*

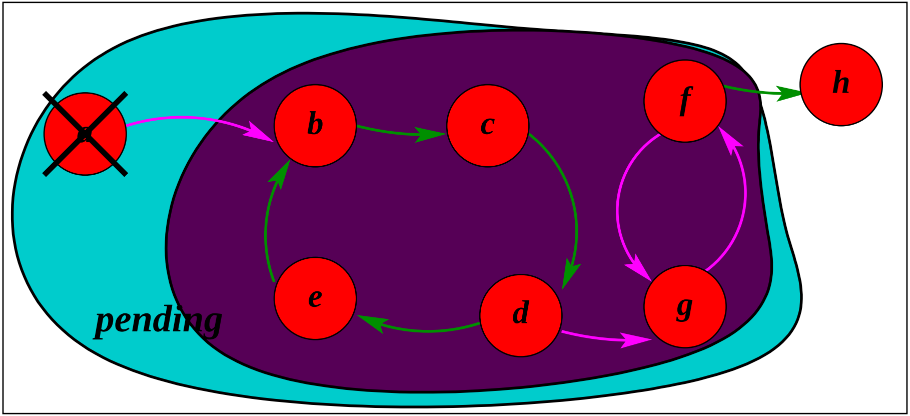


Idea: find unfair states by looking at previous actions within $\langle$*MaybeFair*$\rangle$

Algorithm Example

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

Purple Blob $\equiv$ *MaybeFair*

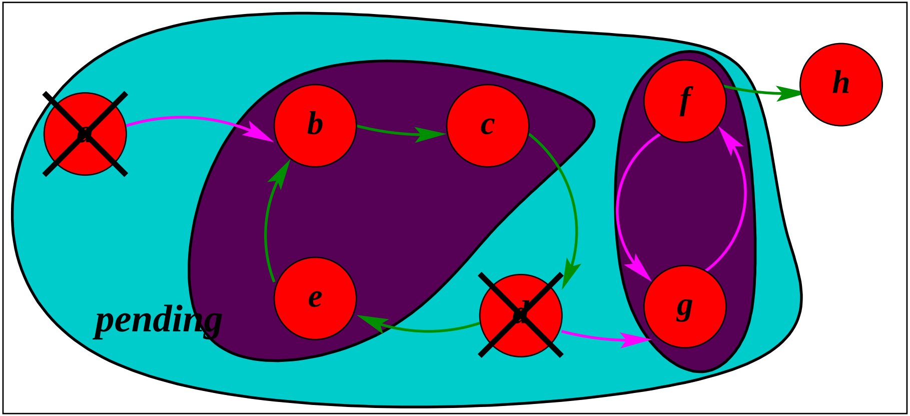


Idea: find unfair states by looking at previous actions within $\langle$*MaybeFair*$\rangle$

Algorithm Example

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

Purple Blob $\equiv$ *MaybeFair*

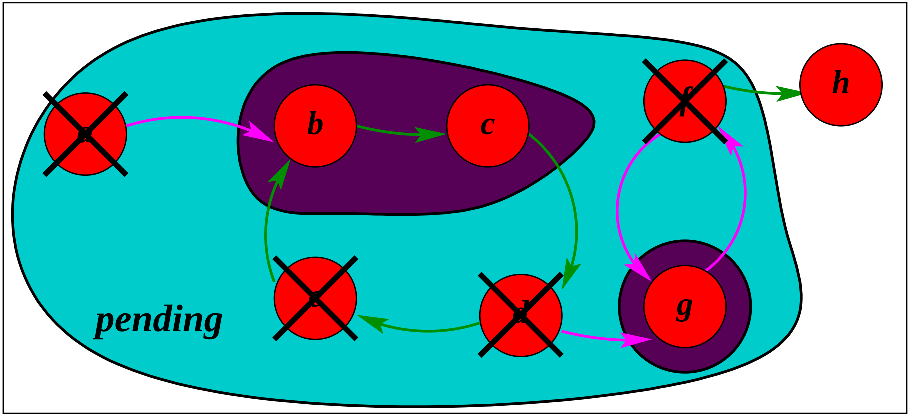


Idea: find unfair states by looking at previous actions within $\langle$*MaybeFair*$\rangle$

Algorithm Example

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

Purple Blob $\equiv$ *MaybeFair*

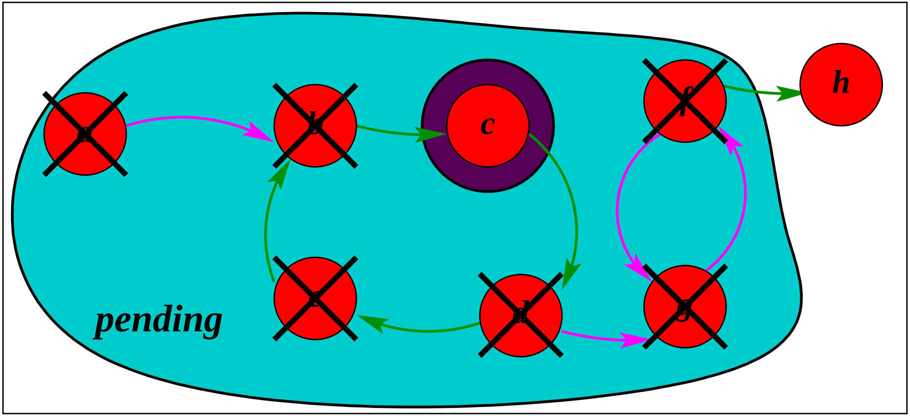


Idea: find unfair states by looking at previous actions within $\langle$*MaybeFair*$\rangle$

Algorithm Example

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

Purple Blob $\equiv$ *MaybeFair*

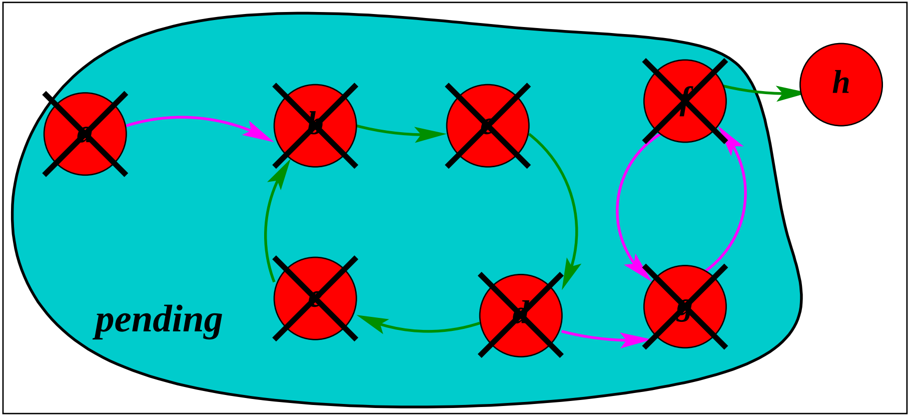


Idea: find unfair states by looking at previous actions within $\langle$*MaybeFair*$\rangle$

Algorithm Example

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

Purple Blob $\equiv$ *MaybeFair*



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

Definition: Predecessor Actions (PAs)

- Suppose $H \subseteq \textit{pending}$. 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 $H \subseteq$ *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 $\langle H \rangle$

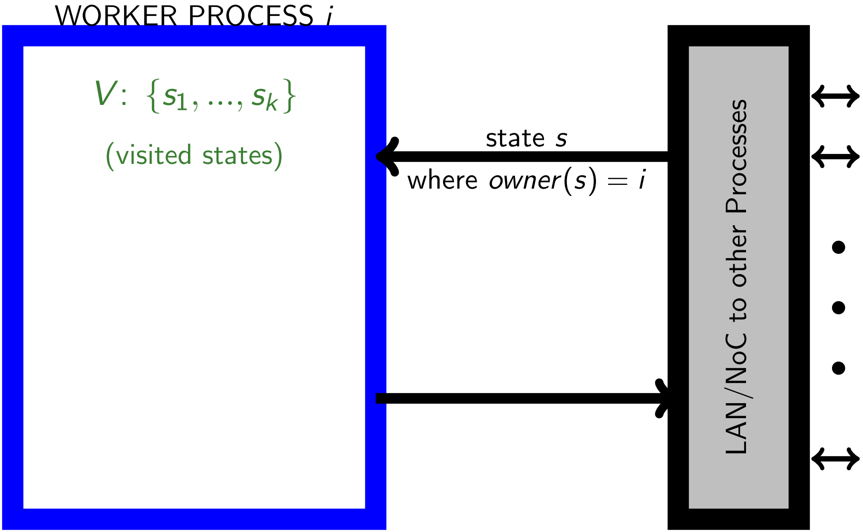
...and $\therefore$ remove s from consideration!

Outline

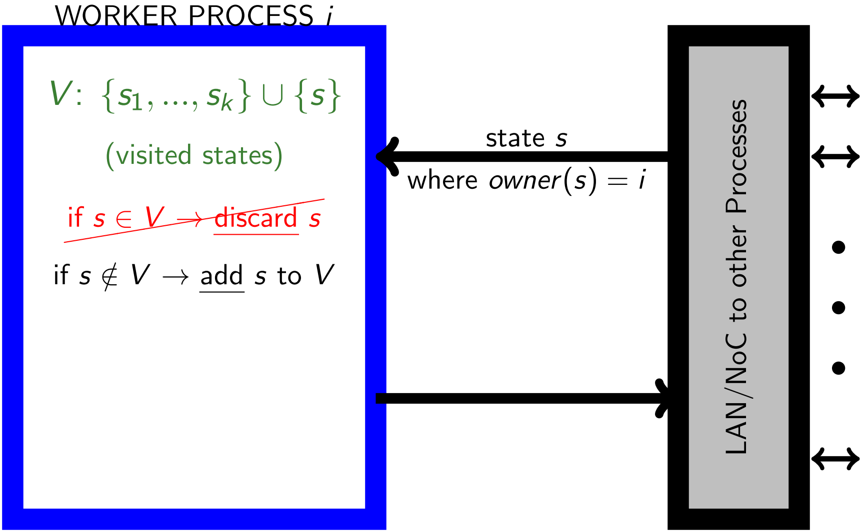
Distributed MC[SD97] Overview

- Simple approach to distributing explicit-state model checking (for safety)
  - Use uniform random hash function $owner : States \rightarrow PIDs$
  - PID i only stores states s such that $owner(s) = i$.
- 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 sucessors are sent to their respective owners
- Termination: when each **WQ** is empty and no messages are in flight

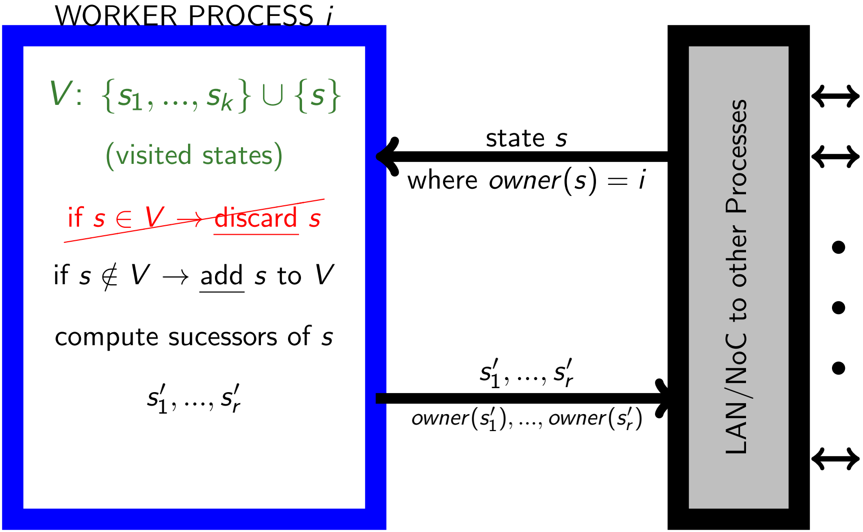
Message Flow



Message Flow



Message Flow



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 $\langle$*MaybeFair*$\rangle$?
- We use $\approx 16 + |\mathcal{C} \cup \mathcal{J}|$ extra bits per state

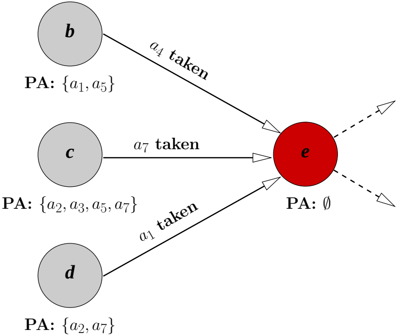
Tracking Predecessor Actions

- Suppose $\mathcal{C} = \{a_1, ...a_k\}$
- "Tag" each hash table entry with PAs, which is a subset of $\mathcal{C}$
  - (plus a few other bookkeeping bits)
- For states in $s \in \mathit{MaybeFair}$: 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 $owner(s')$
  - Receive msg $[s', F]$: $PA(s') := PA(s') \cup F$; expand state s' if $PA(s')$ changed.
  - Continue until no further expansions.

Tracking Predecessor Actions

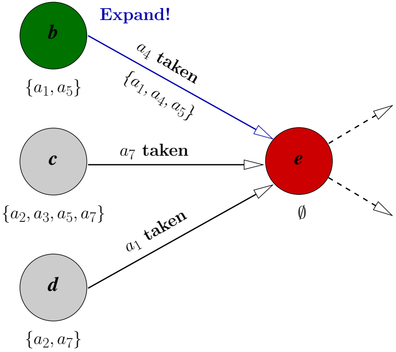
- Suppose $\mathcal{C} = \{a_1, ...a_k\}$
- "Tag" each hash table entry with PAs, which is a subset of $\mathcal{C}$
  - (plus a few other bookkeeping bits)
- For states in $s \in \mathit{MaybeFair}$: initialize $PA(s)$ to $\emptyset$
- Message Passing:
  - Expand state s: if $(s, s') \in a_i$, send msg $[s', PA(s) \cup \{a_i\}]$ to $owner(s')$
  - Receive msg $[s', F]$: $PA(s') := PA(s') \cup F$; expand state s' if $PA(s')$ changed.
  - Continue until no further expansions.
- (A similar idea works for weakly-fair actions)

PA Propagation Example



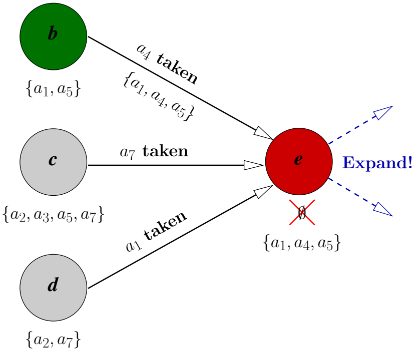
- Strongly-fair actions $\mathcal{C} = \{a_1, ..., a_7\}$

PA Propagation Example



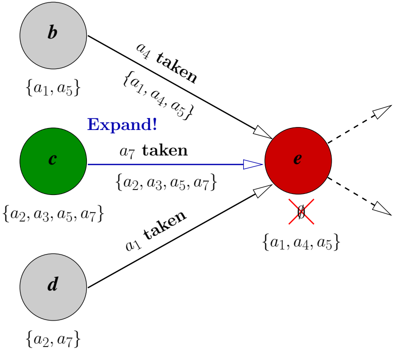
- Strongly-fair actions $\mathcal{C} = \{a_1, ..., a_7\}$

PA Propagation Example



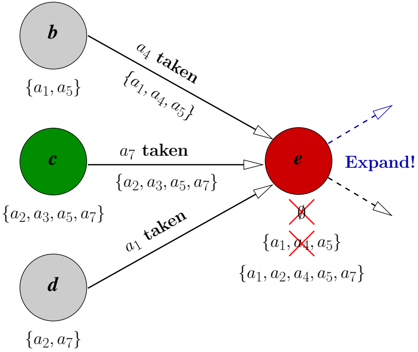
- Strongly-fair actions $\mathcal{C} = \{a_1, ..., a_7\}$

PA Propagation Example



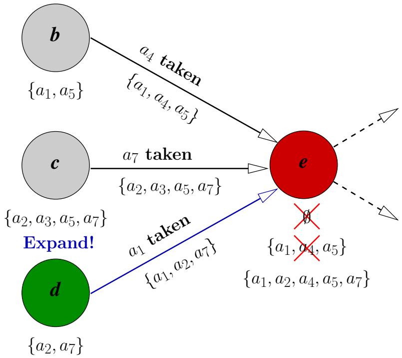
- Strongly-fair actions $\mathcal{C} = \{a_1, ..., a_7\}$

PA Propagation Example



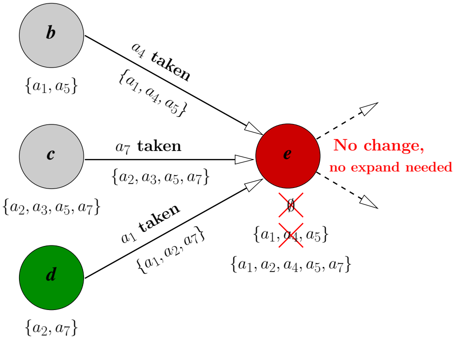
- Strongly-fair actions $\mathcal{C} = \{a_1, ..., a_7\}$

PA Propagation Example



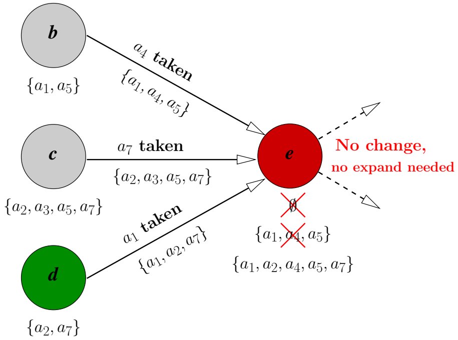
- Strongly-fair actions $\mathcal{C} = \{a_1, ..., a_7\}$

PA Propagation Example



- Strongly-fair actions $\mathcal{C} = \{a_1, ..., a_7\}$

PA Propagation Example



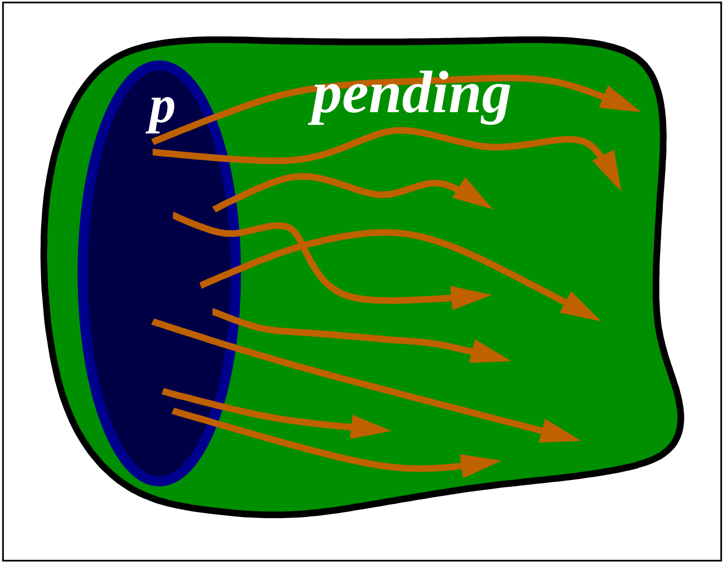
- Strongly-fair actions $\mathcal{C} = \{a_1, ..., a_7\}$
- (once PAs reach a fixpoint, remove unfair states from *MaybeFair*, clear the PAs and compute them again)

Optimization: The "Kernel"

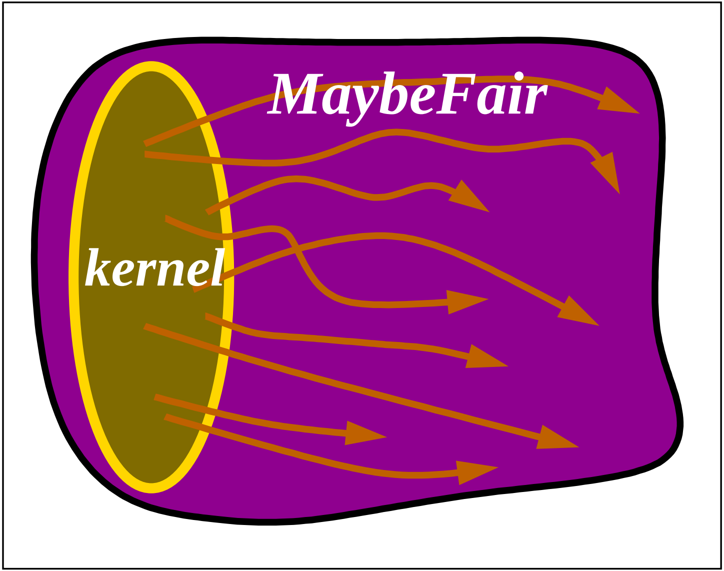
Idea: save set of states K to disk so that *MaybeFair* can be generated through reachability starting with K

- Call K a kernel if $\mathit{MaybeFair} \subseteq \mathit{Reach}(K)$
  - *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 $\mathit{successors}(s) \cap \mathit{MaybeFair}$ into K

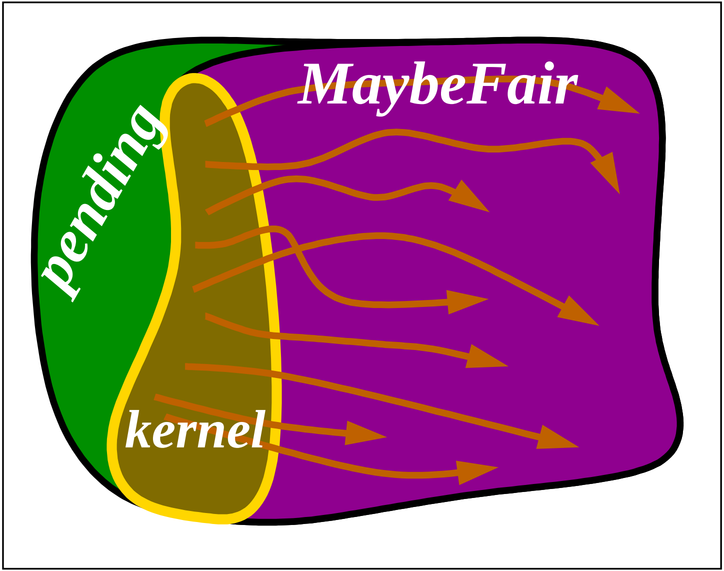
Kernel Optimization



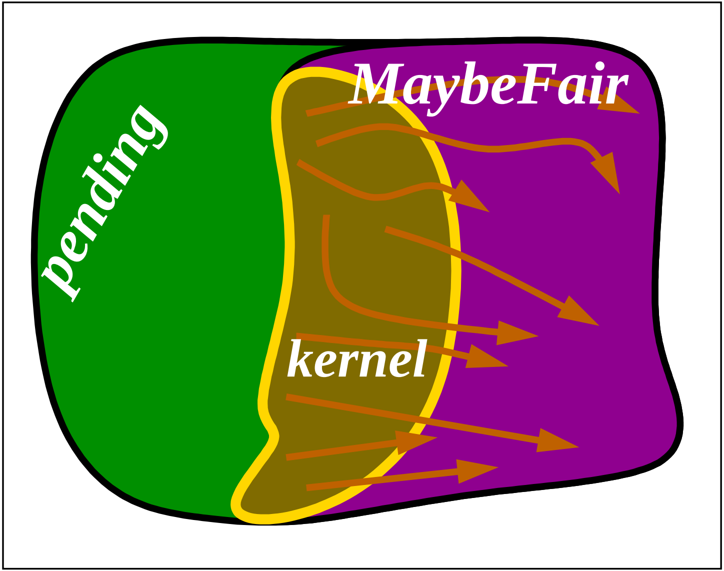
Kernel Optimization



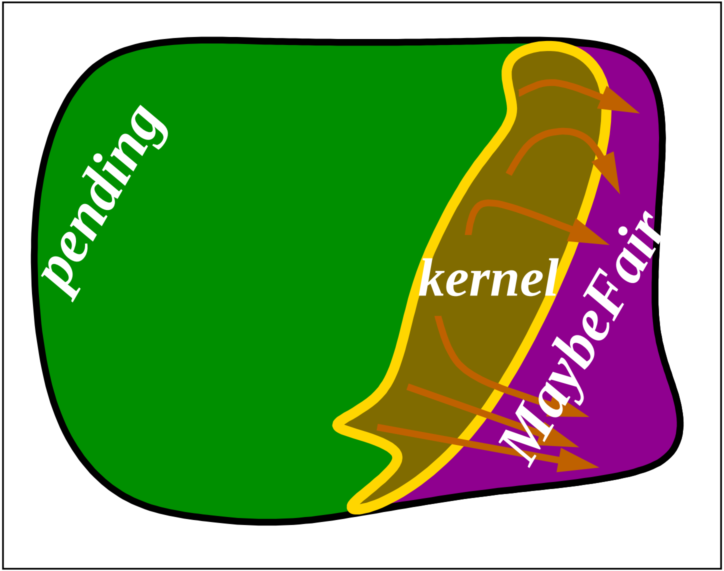
Kernel Optimization



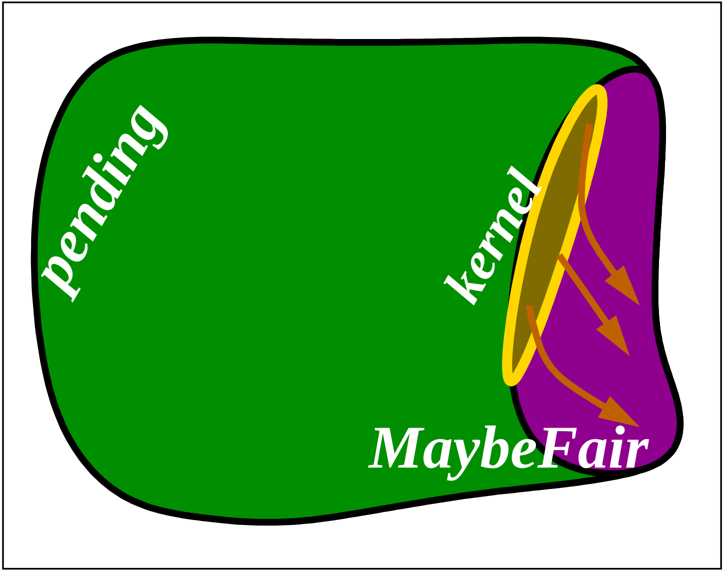
Kernel Optimization



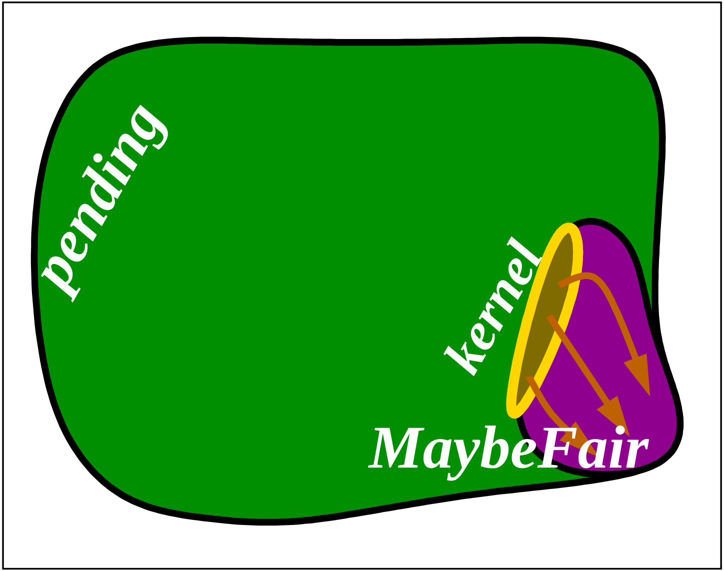
Kernel Optimization



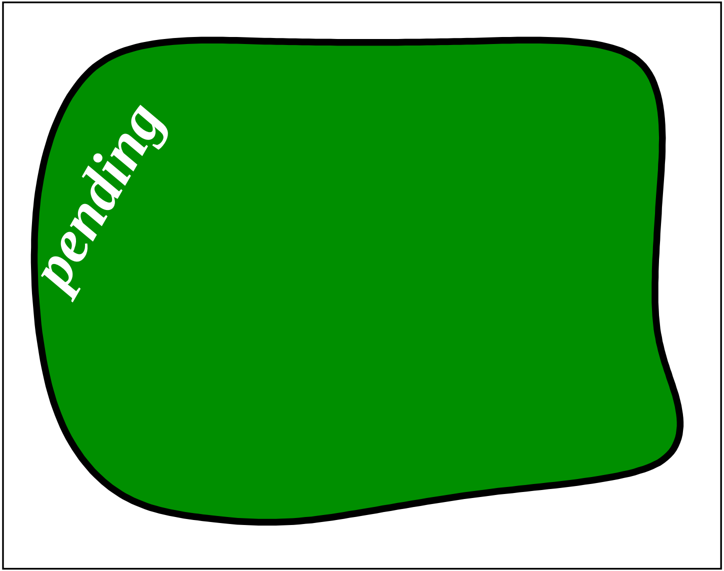
Kernel Optimization



Kernel Optimization



Kernel Optimization



Outline

Performance

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

- *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)

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

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

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!

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! Questions?

References I

U. Stern and D. L. Dill, Parallelizing the murphi verifier, International Conference on Computer Aided Verification, 1997, pp. 256–278.