An $O(m \log n)$ algorithm for stuttering bisimulation

Jeroen Keiren
OU & RU

Joint work with Jan Friso Groote (TU/e), David N. Jansen (RU), and Anton J. Wijs (TU/e)
To appear in ACM Transactions on Computational Logic

27 June 2017

Open Universiteit
www.ou.nl

Today's topic

Improve complexity of deciding stuttering bisimulation equivalence

$$O(m \cdot n) \qquad \longrightarrow \qquad O(m \cdot \log n)$$

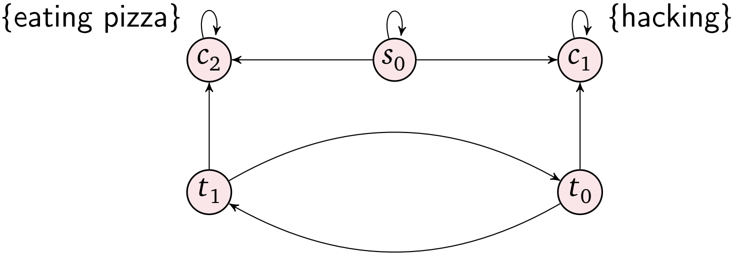
Open Universiteit
www.ou.nl

Kripke structure

$\langle S, AP, \rightarrow, L \rangle$ where:

- S set of states
- $\rightarrow \subseteq S \times S$
- $L : S \rightarrow 2^{AP}$

We let $n = |S|$, $m = |\rightarrow|$.



Strong Bisimulation

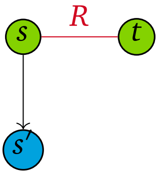
A strong bisimulation is a relation $R \subseteq S \times S$ on the states of a KS $\langle S, AP, \rightarrow, L \rangle$ such that when $s \, R \, t$:

$L(s) = L(t)$, and

Strong Bisimulation

A strong bisimulation is a relation $R \subseteq S \times S$ on the states of a KS $\langle S, AP, \rightarrow, L \rangle$ such that when $s\,R\,t$:

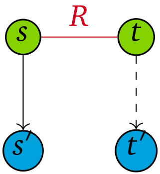
$L(s) = L(t)$, and



Strong Bisimulation

A strong bisimulation is a relation $R \subseteq S \times S$ on the states of a KS $\langle S, AP, \rightarrow, L \rangle$ such that when $s\,R\,t$:

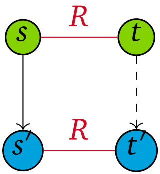
$L(s) = L(t)$, and



Strong Bisimulation

A strong bisimulation is a relation $R \subseteq S \times S$ on the states of a KS $\langle S, AP, \rightarrow, L\rangle$ such that when $s\, R\, t$:

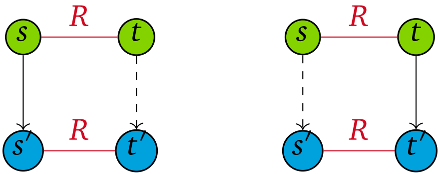
$L(s) = L(t)$, and



Strong Bisimulation

A strong bisimulation is a relation $R \subseteq S \times S$ on the states of a KS $\langle S, AP, \rightarrow, L \rangle$ such that when $s\, R\, t$:

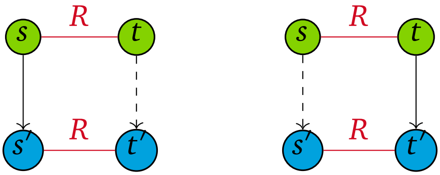
$L(s) = L(t)$, and



Strong Bisimulation

A strong bisimulation is a relation $R \subseteq S \times S$ on the states of a KS $\langle S, AP, \rightarrow, L\rangle$ such that when $s\,R\,t$:

$L(s) = L(t)$, and



States s, t are bisimilar ($s \leftrightarrow t$) iff $s\,R\,t$ for some bisimulation R

Open Universiteit
www.ou.nl

Why strong bisimulation?

- Preserves behaviour
- Allows finding equivalent (simpler) KS
- Preserves truth value of all logical formulas in LTL, CTL, ...

Applications:

- Check implementation conforms to specification
- Interchange specification and implementation of component when reasoning about system
- Reduce model before doing expensive operation (e.g. model checking)

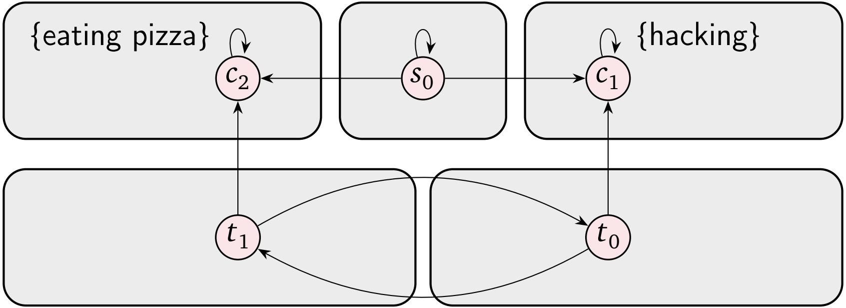
Some terminology

- Equivalence relation (such as strong bisimulation) partitions set of states ...
- ... into disjoint subsets: equivalence classes
- Partition is a cover of S with disjoint subsets
- Disjoint subsets constituting partition are called blocks

Open Universiteit
www.ou.nl

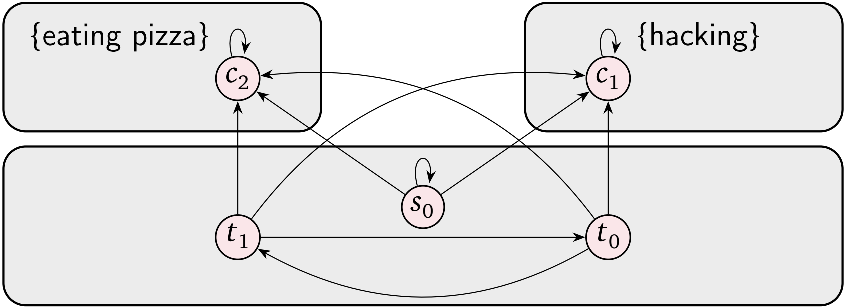
Strong bisimulation

Example



Strong bisimulation

Example



How to compute strong bisimulation?

Partition refinement:

- General technique to approximate equivalence relations from above
- Idea:
  - Start with coarse initial partition
  - Refine blocks until all conditions on equivalence satisfied

Open Universiteit
www.ou.nl

How to compute strong bisimulation?

Partition refinement:

- General technique to approximate equivalence relations from above
- Idea:
  - Start with coarse initial partition
    states with same label are equivalent strong bisimulation condition 1 ✓
  - Refine blocks until all conditions on equivalence satisfied

Open Universiteit
www.ou.nl

How to compute strong bisimulation?

Partition refinement:

- General technique to approximate equivalence relations from above
- Idea:
  - Start with coarse initial partition
    states with same label are equivalent strong bisimulation condition 1 ✓
  - Refine blocks until all conditions on equivalence satisfied
    if $s \rightarrow s'$ and $s \mathrel{R} t$ then $\exists_{t'}\ t \rightarrow t' \ldots$

Open Universiteit
www.ou.nl

How to compute strong bisimulation?

Partition refinement:

- General technique to approximate equivalence relations from above
- Idea:
  - Start with coarse initial partition
    states with same label are equivalent   strong bisimulation condition 1 ✓
  - Refine blocks until all conditions on equivalence satisfied
    if $s \to s'$ and $s \mathrel{R} t$ then $\exists_{t'}\ t \to t' \ldots$

Split the blocks into:

$$split(RfnB, \boldsymbol{SpC}) = \{s \in RfnB \mid \exists_{s' \in S}\ s' \in \boldsymbol{SpC}\}$$
$$cosplit(RfnB, \boldsymbol{SpC}) = RfnB \setminus split(RfnB, \boldsymbol{SpC})$$

Open Universiteit
www.ou.nl

Simple algorithm for strong bisimulation

- Start with coarse initial partition:
  states with same label are equivalent strong bisimulation condition 1 ✓
- Is condition 2 satisfied?
  if $s \rightarrow s'$ and $s\,R\,t$ then $\exists_{t'}\ t \rightarrow t' \ldots$
- If not, block of s' is a splitter. Split between s and t

Split the blocks into:

$$split(RfnB, \boldsymbol{SpC}) = \{s \in RfnB \mid \exists_{s' \in S}\ s' \in \boldsymbol{SpC}\}$$
$$cosplit(RfnB, \boldsymbol{SpC}) = RfnB \setminus split(RfnB, \boldsymbol{SpC})$$

Kanellakis, P.C., Smolka, S.A.: CCS expressions, finite state processes, and three problems of equivalence. Information and Computation. 86, 43–68 (1990).

Algorithm for strong bisimulation

Kanellakis & Smolka

$\mathcal{P} \leftarrow$ initial partition in which states with same label are equivalent
while $\mathcal{P}$ is unstable under some block $SpB \in \mathcal{P}$ **do**
  In $\mathcal{P}$, replace all predecessors of $RfnB$ with two blocks $split(RfnB, SpB)$ and $cosplit(RfnB, SpB)$
end while

Open Universiteit
www.ou.nl

Algorithm for strong bisimulation

Kanellakis & Smolka, with some detail

$\mathcal{P} \leftarrow$ initial partition in which states with same label are equivalent
while $\mathcal{P}$ is unstable **do**
  for $SpB \in \mathcal{P}$ **do** {Find a splitter}
    Mark all predecessors of states in SpB
    if Some predecessor of SpB is in block which is not marked completely
    then {SpB is a splitter}
      for Each marked predecessor block $RfnB$ of SpB **do**
        In $\mathcal{P}$, replace $RfnB$ with two blocks $split(RfnB, SpB)$ and $cosplit(RfnB, SpB)$
      end for
    end if
  end for
end while

Naive algorithm for strong bisimulation

Complexity

- Initialisation . . . . $O(n)$
- Number of splits . . . . $O(n)$
  partitions with 1 state cannot be split
- Finding a splitter . . . . $O(m)$
  every incoming transition traversed at most once
- Splitting w.r.t. a block . . . . $O(m)$
  change blocks and update bookkeeping

Total running time: . . . . $O(mn)$

Efficient refinement step for strong bisimulation

- Maintain coarse partition $\mathcal{C}$ of constellations
  - to store which potential splitters have been checked
- Constellations are unions of blocks
- Constellation in $\mathcal{C}$ is trivial if it corresponds with a single block in $\mathcal{P}$

Maintain the following invariant

- $\mathcal{P}$ is stable w.r.t. each constellation in $\mathcal{C}$

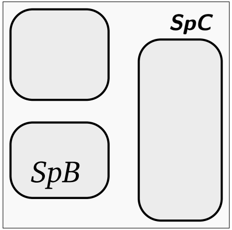
Principle

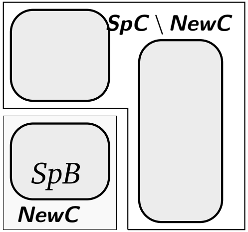
- Process the smaller half (idea from Hopcroft, later Paige & Tarjan)

Paige, R., Tarjan, R.E.: Three Partition Refinement Algorithms SIAM Journal on Computing. 16, 973–989 (1987).

Efficient refinement step for strong bisimulation

...
while $\mathcal{C}$ contains a non-trivial constellation ***SpC*** **do**
  Choose a small splitter block $SpB \subset$ ***SpC*** $\{|SpB| \leq |$***SpC***$/2|\}$
  In $\mathcal{C}$, replace ***SpC*** with ***NewC*** $= SpB$ and ***SpC*** $\setminus$ ***NewC***
  ...
end while
...



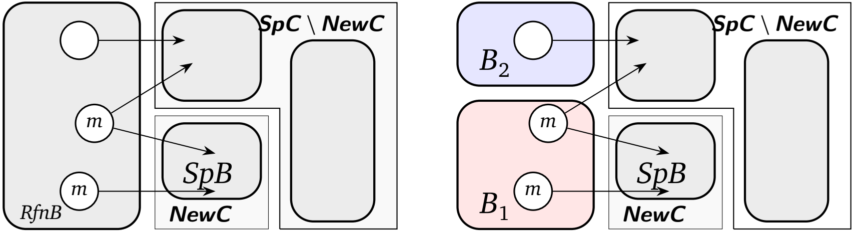


Open Universiteit
www.ou.nl

Efficient refinement step for strong bisimulation

Algorithm

. . .
Mark all predecessors of states in SpB
for each marked predecessor block $RfnB$ of SpB **do**
  $B_1 \leftarrow split(RfnB, SpB)$
  $B_2 \leftarrow cosplit(RfnB, SpB)$ {Stable w.r.t. ***SpC*** \ ***NewC***} . . .
end for



Efficient refinement step for strong bisimulation

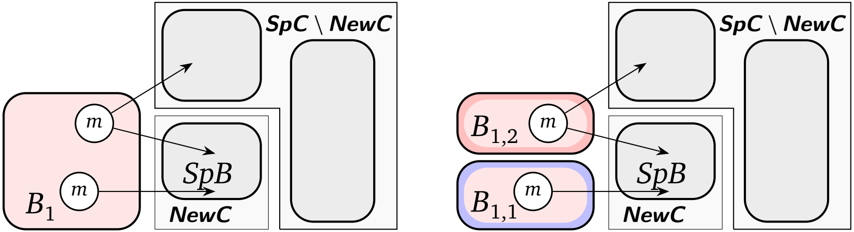
Algorithm

…

$B_{1,1} \leftarrow split(B, \textbf{\textit{SpC}} \setminus \textbf{\textit{NewC}})$

$B_{1,2} \leftarrow cosplit(B, \textbf{\textit{SpC}} \setminus \textbf{\textit{NewC}})$

In $\mathcal{P}$, replace $RfnB$ with three blocks: $B_{1,1}$, $B_{1,2}$, B_2 …



Efficient refinement step for strong bisimulation

Algorithm (Paige & Tarjan)

$\mathcal{P} \leftarrow$ initial partition in which states with same label are equivalent
$\mathcal{C} \leftarrow \{S\}$
while $\mathcal{C}$ contains a non-trivial constellation ***SpC*** **do**
  Choose a small splitter block $SpB \subset \textbf{\textit{SpC}}$ $\{|SpB| \leq |\textbf{\textit{SpC}}/2|\}$
  In $\mathcal{C}$, replace ***SpC*** with $\textbf{\textit{NewC}} = SpB$ and $\textbf{\textit{SpC}} \setminus \textbf{\textit{NewC}}$
  Mark all predecessors of states in SpB
  for each marked predecessor block $RfnB$ of SpB **do**
    $B_1 \leftarrow split(RfnB, SpB)$
    $B_2 \leftarrow cosplit(RfnB, SpB)$ {Stable w.r.t. $\textbf{\textit{SpC}} \setminus \textbf{\textit{NewC}}$}
    $B_{1,1} \leftarrow split(B, \textbf{\textit{SpC}} \setminus \textbf{\textit{NewC}})$
    $B_{1,2} \leftarrow cosplit(B, \textbf{\textit{SpC}} \setminus \textbf{\textit{NewC}})$
    In $\mathcal{P}$, replace $RfnB$ with three blocks: $B_{1,1}$, $B_{1,2}$, B_2
  end for
end while

Open Universiteit
www.ou.nl

Time complexity

- State is in a splitter at most $\lfloor \log_2 n \rfloor$ times
- Every time s is selected, we do at most $O(|in(s)|)$ work
- Time complexity: $\sum_{s \in S} O(|in(s)|) \lfloor \log_2 n \rfloor) = O(m \log n)$

Open Universiteit
www.ou.nl

Stuttering bisimulation

A divergence blind stuttering bisimulation is a relation $R \subseteq S \times S$ so that when $s\,R\,t$:

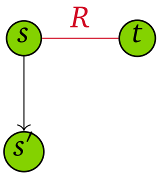
$L(s) = L(t)$, and

Open Universiteit
www.ou.nl

Stuttering bisimulation

A divergence blind stuttering bisimulation is a relation $R \subseteq S \times S$ so that when $s\,R\,t$:

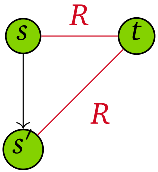
$L(s) = L(t)$, and



Stuttering bisimulation

A divergence blind stuttering bisimulation is a relation $R \subseteq S \times S$ so that when $s \, R \, t$:

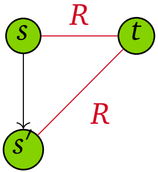
$L(s) = L(t)$, and



Stuttering bisimulation

A divergence blind stuttering bisimulation is a relation $R \subseteq S \times S$ so that when $s \, R \, t$:

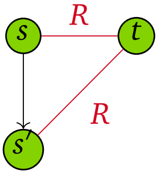
$L(s) = L(t)$, and

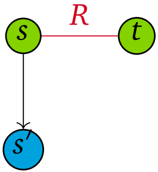


Stuttering bisimulation

A divergence blind stuttering bisimulation is a relation $R \subseteq S \times S$ so that when $s\, R\, t$:

$L(s) = L(t)$, and



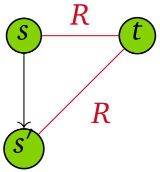


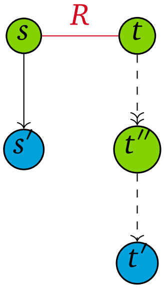
Open Universiteit
www.ou.nl

Stuttering bisimulation

A divergence blind stuttering bisimulation is a relation $R \subseteq S \times S$ so that when $s\,R\,t$:

$L(s) = L(t)$, and

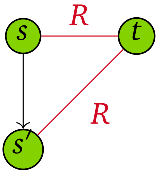


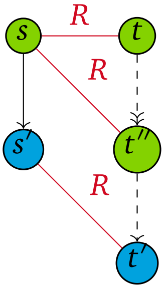


Stuttering bisimulation

A divergence blind stuttering bisimulation is a relation $R \subseteq S \times S$ so that when $s \, R \, t$:

$L(s) = L(t)$, and

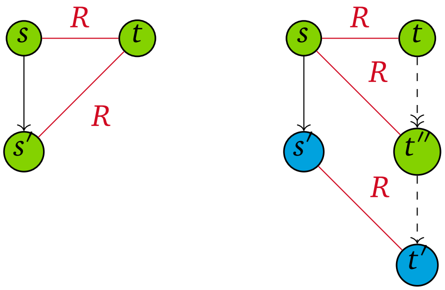




Stuttering bisimulation

A divergence blind stuttering bisimulation is a relation $R \subseteq S \times S$ so that when $s \, R \, t$:

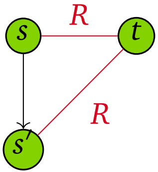
$L(s) = L(t)$, and

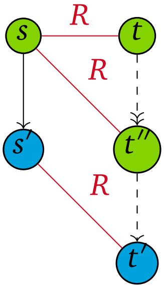


Stuttering bisimulation

A divergence blind stuttering bisimulation is a relation $R \subseteq S \times S$ so that when $s\,R\,t$:

$L(s) = L(t)$, and



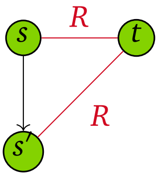


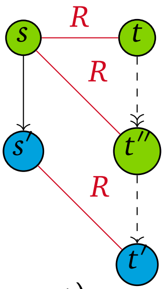
$+$ symmetric cases

Stuttering bisimulation

A divergence blind stuttering bisimulation is a relation $R \subseteq S \times S$ so that when $s \; R \; t$:

$L(s) = L(t)$, and





+ symmetric cases

s, t are divergence blind stuttering bisimilar ($s \leftrightarrow_b t$)
iff $s \; R \; t$ for some divergence blind stuttering bisimulation R

Open Universiteit
www.ou.nl

Why stuttering bisimulation?

- Better reduction than strong bisimulation
- Preserves temporal logics without next-state operator, e.g. LTL$\setminus$X
- Balance between reduction and algorithmic complexity

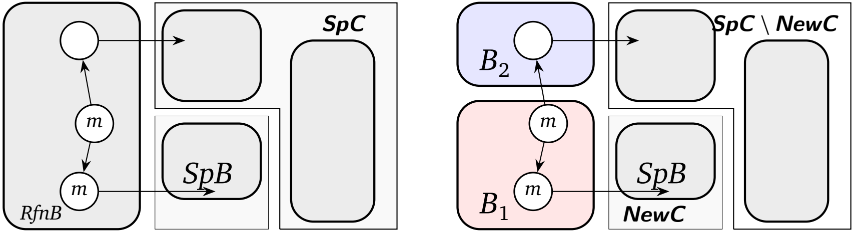
Simple algorithm for stuttering bisimulation

Idea: use algorithm for strong bisimulation, but:

$$\begin{aligned} split(RfnB, \textbf{\textsf{SpC}}) &= \{s \in RfnB \mid \exists_{k \in \mathbb{N}, s_0, \ldots, s_k \in S}\ s = s_0 \\ &\qquad \wedge \forall_{i<k}\ s_i \to s_{i+1} \wedge s_i \in RfnB \wedge s_k \in \textbf{\textsf{SpC}}\} \\ cosplit(RfnB, \textbf{\textsf{SpC}}) &= RfnB \setminus split(RfnB, \textbf{\textsf{SpC}}) \end{aligned}$$

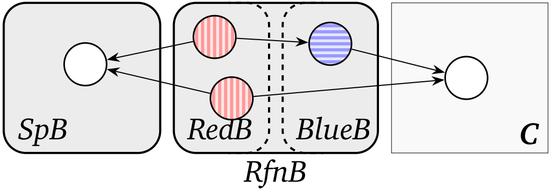
Refinement for stuttering bisimulation

...
Mark all predecessors of states in SpB
Extend marking through inert transitions in SpB
for each marked predecessor block $RfnB$ of SpB **do**
  $B_1 \leftarrow$ marked states in $RfnB$
  $B_2 \leftarrow$ unmarked states in $RfnB\ldots$
end for



Problems with simple algorithm

- Extending marking is inefficient: we visit more than just the marked states, so time complexity of $O(in(SpB)$ not met.
  Solution: Process the smaller half, again, by balancing search for blue/red states
- Invariant not automatically reestablished after splitting.



Solution: Perform additional splits to reestablish invariant

Efficient algorithm

Complete pseudocode

Algorithm 2 Main loop of partition refinement for divergence-blind stuttering equivalence

function DBSTUTTERINGEQUIVALENCE(S, AP, $\rightarrow$, L)
{Find the divergence-blind stuttering equivalence classes for Kripke structure (S, AP, $\rightarrow$, L) with $n \in \mathcal{O}(m)$.}

$\mathcal{P} := \mathcal{P}_0$, i. e. the initial, cycle-free partition; $\mathcal{C} := \{S\}$
Initialise all temporary data — } $\mathcal{O}(m \log n)$

while $\mathcal{C}$ contains a non-trivial constellation ***SpC*** **do** — } $\leq n$ iterations

  Choose a small splitter block $SpB \subset \boldsymbol{SpC}$ from $\mathcal{P}$, i. e. $|SpB| \leq \frac{1}{2}|\boldsymbol{SpC}|$
  Create a new constellation ***NewC*** and move SpB from ***SpC*** to ***NewC***
  $\mathcal{C}$:= partition $\mathcal{C}$ where SpB is removed from ***SpC*** and ***NewC*** is added — } $\mathcal{O}(1)$
  Mark block SpB as refinable
  Mark all states of SpB as predecessors

  for all $s \in SpB$ **do** {Find predecessors of SpB}
    for all $s' \in in(s) \setminus SpB$ **do**
      Mark the block of s' as refinable
      Mark s' as predecessor of SpB
      Register that $s' \rightarrow s$ goes to ***NewC*** (instead of ***SpC***) — } $\mathcal{O}\left(\begin{matrix}|in(SpB)| + \\ |out(SpB)|\end{matrix}\right)$
      Store whether s' still has some transition to $\boldsymbol{SpC} \setminus SpB$
    end for
    Register that inert transitions from s go to ***NewC*** (instead of ***SpC***)
    Store whether s still has some transition to $\boldsymbol{SpC} \setminus SpB$
  end for

  for all refinable blocks $RfnB$ **do** {Stabilise the partition again} — } $\leq |in(SpB)|$ iterations
    Mark block $RfnB$ as non-refinable — } $\mathcal{O}(1)$
    $\langle RedB, BlueB\rangle$:= REFINE($RfnB$, ***NewC***, {marked states $\in RfnB$}, $\emptyset$)
    if $RedB$ contains new bottom states **then**
      $RedB$:= POSTPROCESSNEWBOTTOM($RedB$, $BlueB$)
    end if
    $\langle RedB, BlueB\rangle$:= REFINE($RedB$, $\boldsymbol{SpC} \setminus SpB$, $\emptyset$, {transitions $RedB \rightarrow \boldsymbol{SpC} \setminus SpB$})
    if $RedB$ contains new bottom states **then**
      POSTPROCESSNEWBOTTOM($RedB$, $BlueB$)
    end if
    Unmark all states of the original $RfnB$ as predecessors — } $\mathcal{O}(1)$
  end for
end while
return $\mathcal{P}$

Open Universiteit
www.ou.nl

Efficient algorithm

Complete pseudocode

Algorithm 3 Refine a block under ***SpC***

```
3.1  function Refine(RfnB, SpC, Red, FromRed)
     {Try to refine block RfnB, depending on whether states have (weak) transitions to the splitter con-
     stellation SpC. States in Red are known to have such a transition; alternatively, FromRed contains
     all strong transitions from RfnB to SpC. If FromRed ≠ ∅, then bottom states that are not in Red
     can be tested quickly whether they have such a transition.}
3.2  if RfnB ⊆ SpC then  return ⟨RfnB, ∅⟩
3.3  Test := {bottom states} \ Red,    Blue := ∅                                      } 𝒪(1)
3.4  begin {Spend the same amount of work on either coroutine:}                       }
3.5    whenever |Blue| > ½|RfnB| then       ‖ whenever |Red| > ½|RfnB| then           } 𝒪(1) per assignment
                    Abort this coroutine    ‖               Abort this coroutine      } to Blue or Red, resp.
3.6    while Test ≠ ∅ ∧ FromRed ≠ ∅ do      ‖ while FromRed ≠ ∅ do                    }
3.7      Choose s ∈ Test                    ‖   Choose s → t ∈ FromRed                }
3.8      if s → SpC then                    ‖   Test := Test \ {s}                    }
3.9        Move s from Test to Red          ‖   Red := Red ∪ {s}                      } 𝒪(|Test|)
3.10     else                               ‖   FromRed := FromRed \ {s → t}          } and
3.11       Move s from Test to Blue         ‖                                         } 𝒪(|FromRed|)
3.12     end if                             ‖                                         }
3.13   end while                            ‖ end while                               }
3.14   Blue := Blue ∪ Test                  ‖
3.15   while Blue contains                  ‖ while Red contains                      }
                   unvisited states do      ‖               unvisited states do       }
3.16     Choose an unvisited s ∈ Blue       ‖   Choose an unvisited s ∈ Red           }
3.17     Mark s as visited                  ‖   Mark s as visited                     }    ( |in(NewB)| +    )
3.18     for all s′ ∈ in_τ(s) \ Red do      ‖   for all s′ ∈ in_τ(s) do               }  𝒪 ( |out(NewB)| +   )
3.19       if notblue(s′) undefined then    ‖                                         }    ( |out(NewBott)| )
3.20         notblue(s′) := |out_τ(s′)|     ‖                                         } and
3.21       end if                           ‖                                         } 𝒪(|in(NewB)|)
3.22       notblue(s′) := notblue(s′) − 1   ‖                                         }
3.23       if notblue(s′) = 0 ∧ (FromRed =  ‖                                         }
                   ∅ ∨ s′ ↛ SpC) then       ‖                                         }
3.24         Blue := Blue ∪ {s′}            ‖     Red := Red ∪ {s′}                   }
3.25       end if                           ‖                                         }
3.26     end for                            ‖   end for                               }
3.27   end while                            ‖ end while                               }
3.28   Abort the other coroutine            ‖ Abort the other coroutine               } 𝒪(|out(NewB)|)
3.29   Move Blue to a new block NewB        ‖ Move Red to a new block NewB            }
3.30   Destroy all temporary data           ‖ Destroy all temporary data              } as lines 3.6–3.27
3.31   for all s ∈ NewB do                  ‖ for all non-bottom s ∈ NewB do          }
3.32     for all s′ ∈ in_τ(s) \ NewB do     ‖   for all s′ ∈ out_τ(s) \ NewB do       }
3.33       s′ → s is no longer inert        ‖     s → s′ is no longer inert           } 𝒪(|in(NewB)|)
3.34       if |out_τ(s′)| = 0 then          ‖   end for                               } or
3.35         s′ is a new bottom state       ‖   if |out_τ(s)| = 0 then                } 𝒪(|out(NewB)|)
3.36       end if                           ‖     s is a new bottom state             }
3.37     end for                            ‖   end if                                }
3.38   end for                              ‖ end for                                 }
3.39   RedB := RfnB,    BlueB := NewB       ‖ RedB := NewB,    BlueB := RfnB          } 𝒪(1)
3.40 end                                                                              }
3.41 𝒫 := partition 𝒫 where NewB is added and the states in NewB are removed from RfnB
3.42 return ⟨RedB, BlueB⟩ (with old and new bottom states separated)
```

Open Universiteit
www.ou.nl

Efficient algorithm

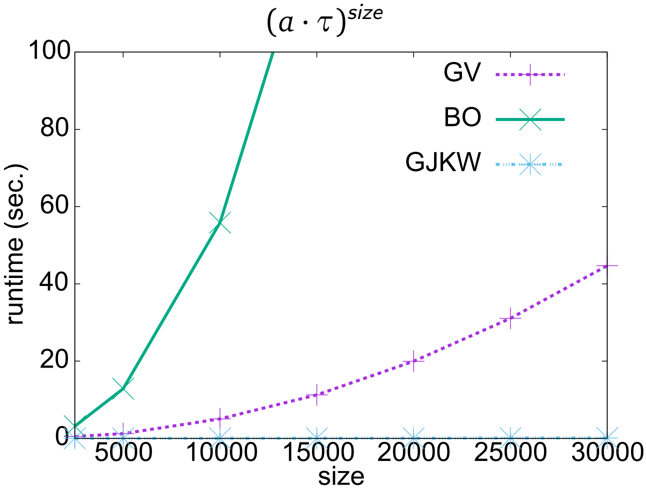
Complete pseudocode

Algorithm 4 Refine as required by new bottom states, called in lines 2.24 and 2.28

4.1 **function** POSTPROCESSNEWBOTTOM(*RedB*, *BlueB*)
{Stabilise the partition for all new bottom states in *RedB*.}
4.2 Create an empty search tree $\mathscr{R}$ of constellations $\quad \mathcal{O}(1)$
4.3 $\langle ResultB, RfnB\rangle :=$ REFINE$(RedB, cosplit(RedB, BlueB), \{$old bottom states $\in RedB\}, \emptyset)$
4.4 **for all** constellations $\boldsymbol{C} \notin \mathscr{R}$ reachable from *RfnB* **do** $\quad \leq |out(NewBott)|$ iter'ns
4.5 Add $\boldsymbol{C}$ to $\mathscr{R}$ $\quad \mathcal{O}(\log n)$
4.6 Register that the transitions $RfnB \to \boldsymbol{C}$ need postprocessing $\quad \mathcal{O}(1)$
4.7 **end for**
4.8 **for all** bottom states $s \in RfnB$ **do**
4.9 Set the current constellation pointer of s to the first constellation it can reach
4.10 **end for** $\quad \mathcal{O}(|NewBott|)$
4.11 **for all** constellations $\boldsymbol{SpC} \in \mathscr{R}$ (in order) **do**
4.12 **for all** blocks $\hat{B}$ with transitions to $\boldsymbol{SpC}$ that need postprocessing **do** $\quad \leq |out(NewBott)|$ iter'ns
4.13 Delete $\hat{B} \to \boldsymbol{SpC}$ from the transitions that need postprocessing $\quad \mathcal{O}(1)$
4.14 $\langle RedB, BlueB\rangle :=$ REFINE$(\hat{B}, \boldsymbol{SpC}, \emptyset, \{$transitions $\hat{B} \to \boldsymbol{SpC}\})$
4.15 **for all** old bottom states $s \in RedB$ **do**
4.16 Advance the current constellation pointer of s to the next constellation it can reach $\quad \mathcal{O}(|out(NewBott) \cap \boldsymbol{SpC}|)$
4.17 **end for**
4.18 **if** *RedB* contains new bottom states **then**
4.19 $\langle \_, RfnB\rangle :=$ REFINE$(RedB, cosplit(RedB, BlueB), \{$old bottom states $\in RedB\}, \emptyset)$
4.20 Register that the transitions $RfnB \to \boldsymbol{SpC}$ need postprocessing $\quad \mathcal{O}(1)$
4.21 Restart the procedure (but keep $\mathscr{R}$), i. e. go to line 4.4
4.22 **end if**
4.23 **end for**
4.24 Delete $\boldsymbol{SpC}$ from $\mathscr{R}$ $\quad \mathcal{O}(\log n)$
4.25 **end for**
4.26 Destroy all temporary data
4.27 **return** *ResultB*

Experimental results

Special case



Open Universiteit
www.ou.nl

Experimental results

Model	original		minimised		running time (in s)		
	n	m	n	m	GV	BO	GJKW
vasy_69_520	69,754	520,633	69,753	520,632	**1.20**	5.00	1.40
vasy_66_1302	66,929	1,302,664	51,128	1,018,692	**2.20**	9.00	3.00
vasy_4338_15666	4,338,672	15,666,588	704,737	3,972,600	1,800.00	300.00	**41.00**
vasy_11026_24660	11,026,932	24,660,513	775,618	2,454,834	1,900.00	1,300.00	**68.00**
lift6-final	6,047,527	26,539,368	1,699	9,870	59.00	270.00	**51.00**
vasy_12323_27667	12,323,703	27,667,803	876,944	2,780,022	2,500.00	1,100.00	**77.00**
vasy_8082_42933	8,082,905	42,933,110	290	680	100.00	450.00	**57.00**
cwi_7838_59101	7,838,608	59,101,007	62,031	470,230	260.00	6,500.00	**160.00**
dining_14	18,378,370	164,329,284	228,486	2,067,856	730.00	2,000.00	**490.00**
cwi_33949_165318	33,949,609	165,318,222	12,463	71,466	620.00	5,600.00	**500.00**
1394-fin3	126,713,623	276,426,688	160,258	538,936	68,000.00	10,000.00	**1,000.00**

Open Universiteit
www.ou.nl

Summary

- Vast improvement: $O(m \log n)$ instead of $O(mn)$
- Fast in practice
- Can also be used for branching bisimulation $O(m(\log |Act| + \log n))$

Open Universiteit
www.ou.nl

Future work

- Branching bisimulation in $O(m \log n)$?

Future work

- Branching bisimulation in $O(m \log n)$?
- Improve governed stuttering bisimulation for parity games (currently $O(mn^2)$)

Open Universiteit
www.ou.nl

Future work

- Branching bisimulation in $O(m \log n)$?
- Improve governed stuttering bisimulation for parity games (currently $O(mn^2)$)
- Investigate impact on other relations such as orthogonal bisimulation

Future work

- Branching bisimulation in $O(m \log n)$?
- Improve governed stuttering bisimulation for parity games (currently $O(mn^2)$)
- Investigate impact on other relations such as orthogonal bisimulation
- Machine-checked proof of running time complexity?

Open Universiteit
www.ou.nl

Thank you

Open Universiteit
www.ou.nl