Stuttering equivalence is too slow!

Stuttering equivalence is too slow!

David N. Jansen[†] Jeroen J. A. Keiren[*,†]
dnjansen@cs.ru.nl Jeroen.Keiren@ou.nl

[*]Open University of the Netherlands, School of Computer Science, Netherlands
[†]Radboud Universiteit, Institute for Computing and Information Sciences, Nijmegen, Netherlands

Abstract

Groote and Wijs recently described an algorithm for deciding stuttering equivalence and branching bisimulation equivalence, acclaimed to run in $\mathcal{O}(m \log n)$ time. Unfortunately, the algorithm does not always meet the acclaimed running time. In this paper, we present two counterexamples where the algorithms uses $\Omega(md)$ time. A third example shows that the correction is not trivial. In order to analyse the problem we present pseudocode of the algorithm, and indicate the time that can be spent on each part of the algorithm in order to meet the desired bound. We also propose fixes to the algorithm such that it indeed runs in $\mathcal{O}(m \log n)$ time.

1 Introduction

It has long been an open problem whether the algorithm by Groote and Vaandrager [3] for computing stuttering equivalence [1] and branching bisimulation [2] was optimal. Recently, Groote and Wijs [4, 5] presented an improvement. They describe an algorithm for deciding stuttering equivalence in time $\mathcal{O}(m \log n)$ and space $\mathcal{O}(m)$, where n is the number of states, and m the number of transitions of the Kripke structure at hand, with $m \geq n$. This is an improvement over the previous running time of $\mathcal{O}(mn)$.

Unfortunately, the algorithm [4] falls short of the stated goal. In this paper we introduce two counterexamples where the algorithm will use more time than $\mathcal{O}(m \log n)$, namely $\Omega(md)$, where d is the maximal outdegree of a state in the Kripke structure.

Since the original description of the algorithm relies heavily on auxiliary data structures and pointers in order to ensure that all information is available quickly when needed, without making the data structures any bigger than strictly necessary, the problem with the algorithm is hard to detect in this original description. We therefore first present our understanding of the algorithm as presented in [4] by giving a high-level description in pseudocode, which leaves out as much of the detailed data structures as possible. This also allows us to assign time budgets to its parts, that are satisfied locally and that together allow us to determine the overall time bound. Subsequently, we identify two problems in the original algorithm, by giving counterexamples that lead to running times higher than the desired bound of $\mathcal{O}(m \log n)$. For each of these problems, we indicate how to fix the algorithm. When fixing the second problem, a further complication arises that needs to be resolved in order to meet the desired bound. Yet, ultimately we can confirm the main result of [4] that stuttering equivalence and branching bisimulation can be computed in $\mathcal{O}(m \log n)$ time and $\mathcal{O}(m)$ space.

Throughout this paper we assume the reader is familiar with the definitions of Kripke structure and (divergence-blind) stuttering equivalence and with the auxiliary notions introduced in [4]. The note is best read while having a copy of [4] within reach for reference. We focus our analysis on deciding stuttering equivalence for Kripke structures. The results carry over directly to the computation of branching bisimulation.

Algorithm 1 The main algorithm for stuttering equivalence. Closely follows [4]

```
1.1  Initialise all temporary data.                                          } $\mathcal{O}(m \log n)$
1.2  while there is a nontrivial constellation do
1.3      Choose a nontrivial constellation $\boldsymbol{SpC}$                 }
         and a splitter $SpB \subset \boldsymbol{SpC}$ that is small (i. e. $|SpB| \le \frac{1}{2}|\boldsymbol{SpC}|$).  } $\mathcal{O}(1)$ per splitter $SpB$
1.4      Create a new constellation $\boldsymbol{NewC}$                       }
         and move $SpB$ from $\boldsymbol{SpC}$ to $\boldsymbol{NewC}$.       }
1.5      for all $s \in SpB$ do {Find predecessors of $SpB$}                  }
1.6          for all $s' \in in(s) \setminus SpB$ do                          }
1.7              Mark the block of $s'$ as refinable.                         }
1.8              Mark $s'$ as predecessor of $SpB$.                           } $\mathcal{O}(|in(SpB)|)$
1.9              Register that $s' \to s$ goes to $\boldsymbol{NewC}$ (instead of $\boldsymbol{SpC}$).  }
1.10         end for                                                          }
1.11     end for                                                              }
1.12     Prepare $SpB$ to be refined (i. e., mark states, register transitions).  } $\mathcal{O}(|out(SpB)|)$
1.13     begin {Stabilise the partition again:}
1.14         for all refinable blocks $RfnB$ do
1.15             $Result := \textsc{TrySplit}(RfnB, \boldsymbol{NewC}$, marked states $\in RfnB$,
                                        unmarked bottom states $\in RfnB)$
1.16             $\textsc{TrySplit}'(Result, \boldsymbol{SpC}$, states $\in Result$ with a transition to $\boldsymbol{SpC}$,
                                bottom states $\in Result$ without transition to $\boldsymbol{SpC})$
1.17             $\textsc{PostprocessNewBottom}()$
1.18         end for
1.19     end
1.20     Destroy all temporary data (i. e., markings of states and blocks).   } $\mathcal{O}(|in(SpB)|+|out(SpB)|)$
1.21 end while
```

2 Pseudocode for Groote/Wijs 2016

We first present the main part of the algorithm from [4] as we understand it in terms of pseudocode, while separating out a routine TrySplit that tries to split a refinable block into states that can reach the splitter (called red states) and those that cannot (called blue states). There is a small difference between the calls to TrySplit in lines 1.15 and 1.16, which we will explain later. The high-level structure of the algorithm is presented in Algorithm 1. It maintains the invariant:

Invariant 1. *The current blocks are stable with respect to the constellations, i. e. all states in a block can reach the same constellations through a (weak) transition.*

A nontrivial constellation, i. e. a constellation containing multiple blocks, indicates that the current blocks are not yet stable with respect to themselves. The main loop separates a block SpB from a nontrivial constellation $\boldsymbol{SpC}$, moving it to a new constellation $\boldsymbol{NewC}$, and then restores the invariant by refining blocks with respect to these new constellations, if needed. The latter is done in TrySplit.

In the algorithms we assigned a time budget to some steps to facilitate the analysis of the algorithms' complexity. We generally use the abbreviations $|in(B)| = \sum_{s \in B} \max\{1, |in(s)|\}$ and $|out(B)| = \sum_{s \in B} \max\{1, |out(s)|\}$. Similarly, $in_\tau(s)$ are the *inert* incoming transitions and $out_\tau(s)$ the *inert* outgoing transitions of s.

2.1 Splitting blocks

The most important new idea from [4] is used in the step when a block is actually being refined. They start to find both the red and the blue states, spending the same amount of work on either part, until it becomes clear which one is the smaller. In other words, they use the idea "Process the smaller half" not only when looking for splitters, but also when refining blocks. The work

Algorithm 2 Refine a block into red and blue states, called in Line 1.15. Slightly improved

```
2.1  function TrySplit($RfnB$, $\boldsymbol{SpC}$, $Red$, $Blue$)
2.2  {Try to refine block $RfnB$, depending on whether states have transitions to the splitter constellation $\boldsymbol{SpC}$. $Red$ contains all states in $RfnB$ with a strong transition to $\boldsymbol{SpC}$, and $Blue$ contains all bottom states in $RfnB$ without transition to $\boldsymbol{SpC}$.}
2.3  begin {Spend the same amount of work on either process:}
2.4    whenever $|Blue| > \frac{1}{2}|RfnB|$ then       ||  whenever $|Red|>\frac{1}{2}|RfnB|$ then        } $\mathcal{O}(1)$ per assign-
2.5      Stop this process.                            ||    Stop this process.                         } ment to $Blue$ or
2.6    end whenever                                    ||  end whenever                                 } $Red$, respectively.
2.7    while $Blue$ contains                           ||  while $Red$ contains                         }
                            unvisited states do        ||                    unvisited states do        }
2.8      Choose an unvisited $s \in Blue$.             ||    Choose an unvisited $s \in Red$.           }
2.9      Mark $s$ as visited.                          ||    Mark $s$ as visited.                       }
2.10     for all $s' \in in_\tau(s) \setminus Red$ do  ||    for all $s' \in in_\tau(s)$ do             }
2.11       if $notblue(s')$ undefined then             ||                                               }
2.12         $notblue(s') := |out_\tau(s')|$           ||                                               } $\mathcal{O}(|in_\tau(NewB)|)$
2.13       end if                                      ||                                               }
2.14       $notblue(s') := notblue(s') - 1$            ||                                               }
2.15       if $notblue(s') = 0$ then                   ||                                               }
2.16         $Blue := Blue \cup \{s'\}$                ||      $Red := Red \cup \{s'\}$                 }
2.17       end if                                      ||                                               }
2.18     end for                                       ||    end for                                    }
2.19   end while                                       ||  end while                                    }
2.20   Stop the other process.                         ||  Stop the other process.                      } $\mathcal{O}(|out(NewB)|)$
2.21   Move $Blue$ to a new block $NewB$.              ||  Move $Red$ to a new block $NewB$.            }
2.22   Destroy all temporary data.                     ||  Destroy all temporary data.                  } as Lines 2.7–2.19
2.23   for all $s \in NewB$ do                         ||  for all $s \in NewB$ do                      }
2.24     for all $s' \in in_\tau(s) \setminus NewB$ do ||    for all $s' \in out_\tau(s) \setminus NewB$ do }
2.25       $s' \to s$ is no longer inert.              ||      $s \to s'$ is no longer inert.           } $\mathcal{O}(|in_\tau(NewB)|)$
2.26       if $|out_\tau(s')| = 0$ then                ||    end for                                    }
2.27         $s'$ is a new bottom state.               ||    if $|out_\tau(s)| = 0$ then                } or
2.28       end if                                      ||      $s$ is a new bottom state.               }
2.29     end for                                       ||    end if                                     } $\mathcal{O}(|out_\tau(NewB)|)$
2.30   end for                                         ||  end for                                      }
2.31   $Result := RfnB$                                ||  $Result := NewB$
2.32 end
2.33 return $Result$
```

Algorithm 3 Refine as required by new bottom states, called in Line 1.17

```
3.1  function PostprocessNewBottom()
3.2  for all constellations $\boldsymbol{C}$ reachable from $\hat{B}$ do              } $\mathcal{O}(|out(s)|)$ for some old
3.3      Register that the pair $(\hat{B}, \boldsymbol{C})$ needs postprocessing.      } bottom state $s \in \hat{B}$
3.4  end for
3.5  while there is a pair $(\hat{B}, \boldsymbol{C})$ that needs postprocessing do
3.6      Choose a pair $(\hat{B}, \boldsymbol{C})$ that needs postprocessing.          }
3.7      Delete $(\hat{B}, \boldsymbol{C})$ from the pairs that need postprocessing.  } $\mathcal{O}(1)$ per pair $(\hat{B}, \boldsymbol{C})$
3.8      if not all new bottom states can reach $\boldsymbol{C}$ then                 }
3.9          TrySplit′$(\hat{B}, \boldsymbol{C}$, states $\in \hat{B}$ with a transition to $\boldsymbol{C}$,
                     new bottom states $\in \hat{B}$ without transition to $\boldsymbol{C})$
3.10         for all constellations $\boldsymbol{C'}$ that $NewB$ can reach[a] do      }
3.11             if $(\hat{B}, \boldsymbol{C'})$ still needs postprocessing then       }
3.12                 Register that $(NewB, \boldsymbol{C'})$ needs postprocessing.     } $\mathcal{O}(|out(NewB)|)$
3.13             end if                                                              }
3.14         end for                                                                 }
3.15     end if
3.16 end while
3.17 Destroy all temporary data.
3.18 return
```

[a] *NewB* is the new block created in TrySplit′, Line 3.9.

spent on the refinement can then be bounded: state s is involved in such a refinement at most $\mathcal{O}(\log n)$ times. Upon every such refinement, at most $\mathcal{O}(|in(s)| + |out(s)|)$ time is spent for state s. So, overall $\mathcal{O}((|in(s)| + |out(s)|) \log n)$ time is spent on state s. Summing over all states then gives the desired time complexity $\mathcal{O}(m \log n)$.

Note that this may require detailed bookkeeping of the amount of work. One may balance the work by using an auxiliary variable, which stores the amount of work done on the red states minus the amount of work done on the blue states. Every time a state or transition is checked (i. e. every time the loop in Lines 2.7 or 2.10 is entered), the balance is increased or reduced by 1.

We deviate from [4] slightly: *ibid.* uses a priority queue to keep track of *notblue*, but actually nothing queue-like is needed, as the order of states is irrelevant for the correctness, time and memory bounds. Note that the data structure should allow to test for membership in time $\mathcal{O}(1)$.[1] They use the priority only to store the value $notblue(s')$ and check whether this value is defined by a test for membership in the priority queue. We propose instead to set $notblue(s')$ to some special value (e. g. 0) to indicate that it was not yet calculated. We also need a list or set of all states whose $notblue(s')$ is defined, to destroy the temporary data later.

2.2 New bottom states

While refining a block, it may happen that some states become bottom states because all their inert transitions become transitions from a red state to a blue state and therefore are no longer inert. We have to single out these new bottom states because the algorithm treats them differently from the non-bottom states. Here, we also added a slight improvement (Lines 2.23–2.30): we only look for new bottom states in *Red*.

Additionally, it may happen that a new bottom state can no longer reach all the constellations that were reachable from the original bottom states. To repair Invariant 1, we may have to split some new bottom states off the block. This further splitting itself is shown in Algorithm 3. The basic idea is to check, for each constellation that is reachable from some new bottom state, whether the block has to be split. Of course, as soon as a block is split, both parts have to be checked for the remaining reachable constellations. The algorithm in [4] constructs, for each block $\hat{B}$ and

[1] Note that priority queues typically have longer access times.

constellation $\boldsymbol{C}$, a list $S_{\boldsymbol{C}} \subseteq \hat{B}$ of states that can reach $\boldsymbol{C}$ (see Line 3.3). These lists are used in Line 3.9 to decide which states are blue, namely the new bottom states that are *not* in $S_{\boldsymbol{C}}$. The time budget $\mathcal{O}(|\mathit{out}(\mathit{NewBott})|)$ can be met if the list $S_{\boldsymbol{C}}$ follows the same order as the list *NewBott*.

Algorithm 3 generally follows [4], except in Lines 3.10–3.14, where we tried to find a formulation that fits in the time budget.

3 Counterexamples to time complexity

The algorithm contains two problems that cause it to be too slow. First, the second call to TrySplit, here referred to as TrySplit′, takes too long. We give a counterexample, and improve TrySplit′ to improve the complexity to the required bound. Also the postprocessing of new bottom states as carried out in [4] is too slow. In section 3.3 we analyse the problem in the original algorithm; the proposed improvement has already been incorporated in Algorithm 3.

3.1 TrySplit′ is too slow

In the call to TrySplit′ (in Line 1.16), the initial set of red states is given implicitly, through a list of transitions. As a consequence, the test whether the potentially blue state s' has a non-inert transition to $\boldsymbol{SpC}$ (this is why we require $s' \notin \mathit{Red}$ in Line 2.10ℓ) in the variant TrySplit′ is executed in a different way compared to the one in TrySplit. Groote and Wijs [4] add this test just before Line 2.12ℓ. If s' is marked (i. e. it has a transition to *SpB*), they can access one of their many auxiliary variables, but otherwise, "it can be checked by walking over the transitions $s' \to s'' \in s'.T_{tgt}$" (obviously, this is meant instead of the original "$\ldots \in s.T_{tgt}$" – see Section 5.3, item 1.(b).ii.A.second bullet.first dash of [4]). So they execute a loop to verify $\mathit{out}_{\not\tau}(s') \cap \boldsymbol{SpC} = \emptyset$, namely:

```
for all $s'' \in \mathit{out}(s')$ do
    if $s'' \in \boldsymbol{SpC} \setminus \mathit{RfnB}$ then continue to Line 2.10$\ell$
end for
```

This loop makes their algorithm slower than promised: the test uses time $\mathcal{O}(|\mathit{out}(s')|)$, but should take at most $\mathcal{O}(1)$. Our first counterexample illustrates this time budget overrun.

Assume that the partition shown in Figure 1 has been reached. Then, we refine $\boldsymbol{SpC}$: we select *SpB*, find its weak predecessors (the whole block *RfnB*, so nothing is refined) and the weak predecessors of $\boldsymbol{SpC} \setminus \mathit{SpB}$ (the right half of block *RfnB*). Note that we do the latter without walking over the states in $\boldsymbol{SpC} \setminus \mathit{SpB}$: it is ok to spend $\mathcal{O}(|\mathit{in}(\mathit{SpB})|)$ time here. We also save time by calculating the *complement* of the weak predecessors, i. e. the 🟦-states in *NewB*, because it is smaller than $\mathit{RfnB} \setminus \mathit{NewB}$: we are allowed to spend an additional $\mathcal{O}(|\mathit{in}(\mathit{NewB})| + |\mathit{out}(\mathit{NewB})|)$ time on this task.

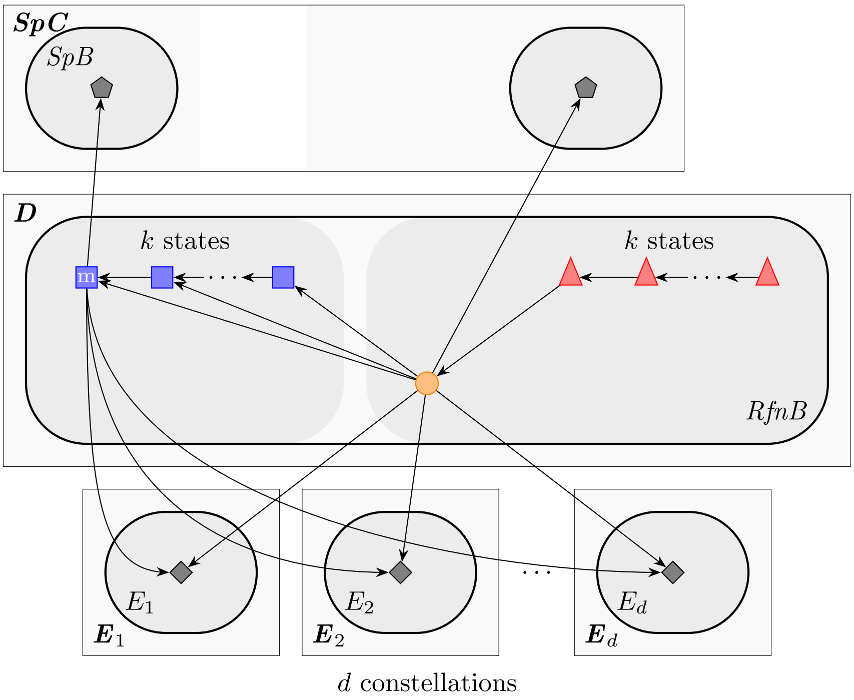
As the algorithm looks through the predecessors of all 🟦-states, it considers the 🟠-state for inclusion in *NewB*. This happens k times (once for each transition to a 🟦-state). The cited passage then requires that we check each time whether some immediate successor of a 🟠-state is in $\boldsymbol{SpC} \setminus \mathit{SpB}$. If we check the transitions to the ◆-states before the ⬟-state, we spend $\Omega(d)$ time, and in the end we find that the 🟠-state should not be considered further.[2] The problem is that we spend (much) time on a transition to *NewB* from a state that is *possibly* in *NewB* but in the end not *actually* in *NewB*. The part of the algorithm in Section 5.3, 1.(b).ii.A.second bullet, is only allowed to spend $\mathcal{O}(1)$ time on each such transition.

Overall, the number of states and transitions both are $\mathcal{O}(k+d)$. So, the algorithm is allowed to spend $\mathcal{O}((k+d)\log(k+d))$ time, but it spends $\Omega(kd)$ time. Variants of this Kripke structure with several copies of the 🟠-state show that the checks can cost $\mathcal{O}(md)$ time.

Note that we are not allowed to concentrate on the red states (the weak predecessors of $\boldsymbol{SpC} \setminus \mathit{SpB}$, the 🔺-states and 🟠-state) themselves instead of the complement, as this set is larger.

[2] In particular, [4] does not define *notblue*(🟠). An ad-hoc solution would be to set it to some value $> d$, but that does not help if there are multiple copies of 🟠.

Figure 1: TrySplit′ is too slow.



Lemma 2. *Refining in Lines 1.16 and 3.9 as described in [4] has a worst-case time complexity of* $\Omega(md)$.

We tried (in vain) to find a recursive counterexample, which should increase the time complexity to $\Omega(md \log n)$, but every of our ideas led to a counterexample with so many additional transitions that it still fit the bound of Lemma 2.

3.2 A faster TrySplit′

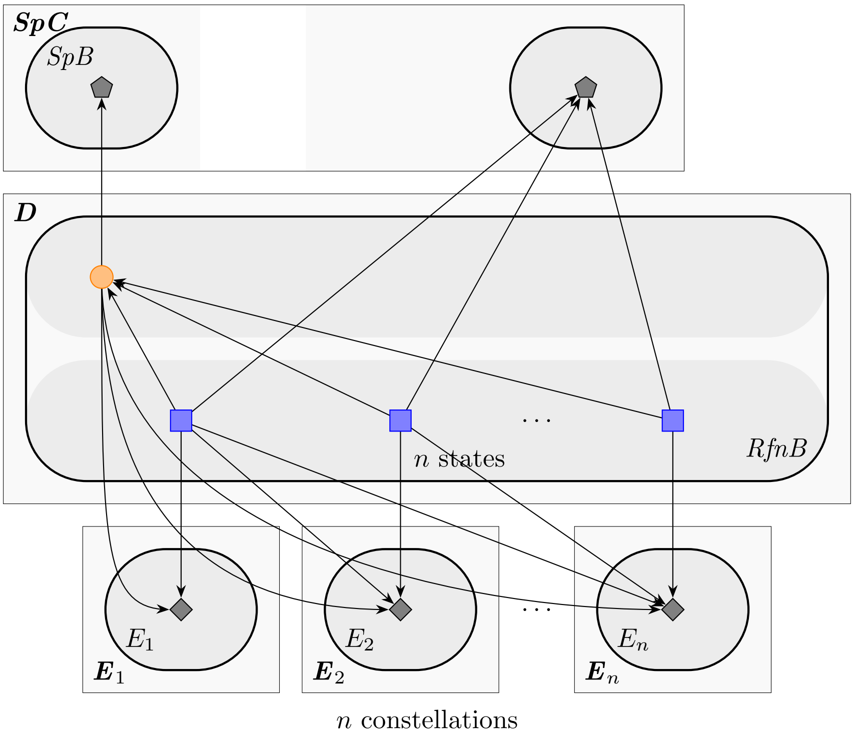
We propose to solve this problem as follows: *Execute the slow test at the latest possible moment,* namely immediately before a state is inserted in *Blue*. This is shown in Algorithm 4. Here, we also present the formal parameter list according to the implicit representation of the red and blue states: a set *FromRed* of transitions from red states, a set *MaybeBlue* of possibly blue bottom states with a predicate *isBlueTest* that indicates which bottom states are actually blue. The overall time budget is still met: If the red states are the smaller part, then *FromRed* is a subset of $out(NewB)$. If the blue states are the smaller part, then states in $MaybeBlue \setminus NewB$ are marked (i. e. they have a transition to *SpB*, see Line 1.8); we are allowed to walk over them one more time.

When the test is executed in Line 4.24ℓ, all inert transitions of s' point to blue states. If s' has a non-inert transition to $\boldsymbol{SpC}$, it is actually a red state, and in particular, a red new bottom state. (It may happen that we do not find all new bottom states here; therefore, we still have to execute Line 4.33ℓ.) As every state becomes a bottom state at most once, we are allowed to spend time $\mathcal{O}((|in(s')| + |out(s')|) \log n)$ then, which is abundant. If no such transition to *SpB* is found, s' is a blue state and we have to account for the time differently. It is $\mathcal{O}(|out(s')|)$ per unmarked blue state s'. Every time s' becomes an unmarked blue state, the test is executed exactly once, which fits in the general bound per time that s' is involved in a refinement.

Algorithm 4 Refine a block w. r. t. $\boldsymbol{SpC} \setminus SpB$ (corrected), called in Line 1.16

```
4.1  function TrySplit′($RfnB$, $\boldsymbol{SpC}$, $FromRed$, $MaybeBlue$, $isBlueTest$)
4.2  {Try to refine block $RfnB$, depending on whether states have transitions to the splitter constellation $\boldsymbol{SpC}$. $FromRed$ contains all transitions from $RfnB$ to $\boldsymbol{SpC}$, $MaybeBlue$ contains all bottom states that may be initially blue states, $isBlueTest$ is a predicate that determines whether a candidate in $MaybeBlue$ is definitely blue.}
4.3  begin {Spend the same amount of work on either process:}
4.4    $Blue := \emptyset$                               ‖ $Red := \emptyset$                                    } $\mathcal{O}(1)$
4.5    whenever $|Blue| > \frac{1}{2}|RfnB|$ then       ‖ whenever $|Red|>\frac{1}{2}|RfnB|$ then             } $\mathcal{O}(1)$ per assign-
4.6      Stop this process.                             ‖   Stop this process.                                  } ment to $Blue$ or
4.7    end whenever                                     ‖ end whenever                                          } $Red$, respectively.
4.8    while $Blue$ or $MaybeBlue$ contain              ‖ while $Red$ or $FromRed$ contain                      }
                          unvisited states do           ‖               unvisited elements do                   }
4.9      Choose an unvisited $s \in Blue$               ‖   Choose an unvisited $s \in Red$                     }
                        or $s \in MaybeBlue$.           ‖          or $s \to s'' \in FromRed$.                  }
4.10     Mark $s$ as visited.                           ‖   if $s \to s''$ is chosen then                       }
4.11     if $s \notin Blue$ then                        ‖     Mark $s \to s''$ as visited.                      }
4.12       if $\neg isBlueTest(s)$ then                 ‖     if $s$ was visited earlier then                   }    ( $|in_\tau(NewB)|+$    )
4.13         continue to Line 4.8$\ell$                 ‖       continue to Line 4.8$r$                         }    ( $|MaybeBlue|+$        )
4.14       end if                                       ‖     end if                                            } $\mathcal{O}$ ( $|out(NewB)|+$   )
4.15       $Blue := Blue \cup \{s\}$                    ‖     $Red := Red \cup \{s\}$                           }    ( $|out(NewBott)|$      )
4.16     end if                                         ‖   end if                                              }
4.17                                                    ‖   Mark $s$ as visited.                                } and
4.18     for all $s' \in in_\tau(s) \setminus Red$ do   ‖   for all $s' \in in_\tau(s)$ do                      } $\mathcal{O}$ ( $|in_\tau(NewB)|+$ )
4.19       if $notblue(s')$ undefined then              ‖                                                       }    ( $|FromRed|$       )
4.20         $notblue(s') := |out_\tau(s')|$            ‖                                                       }
4.21       end if                                       ‖                                                       }
4.22       $notblue(s') := notblue(s')-1$               ‖                                                       }
4.23       if $notblue(s') = 0$ then                    ‖                                                       }
4.24         if $out_{\not\tau}(s') \cap \boldsymbol{SpC} = \emptyset$ then ‖                                   }
4.25           $Blue := Blue \cup \{s'\}$               ‖     $Red := Red \cup \{s'\}$                          }
4.26         end if                                     ‖                                                       }
4.27       end if                                       ‖                                                       }
4.28     end for                                        ‖   end for                                             }
4.29   end while                                        ‖ end while                                             }
4.30   Stop the other process.                          ‖ Stop the other process.                               } $\mathcal{O}(|out(NewB)|)$
4.31   Move $Blue$ to a new block $NewB$.               ‖ Move $Red$ to a new block $NewB$.                     }
4.32   Destroy all temporary data.                      ‖ Destroy all temporary data.                           } as Lines 4.8–4.29
4.33   Find new non-inert transitions and bottom states (as Lines 2.23–2.30).
4.34 end
4.35 return
```

Figure 2: PostprocessNewBottom is too slow.



3.3 PostprocessNewBottom is too slow

In Algorithm 3, we already included an improvement in Lines 3.10–3.14. If some block $\hat{B}$ is split here, it should not take longer than $\mathcal{O}\left(|in(NewB)| + |out(NewB)|\right)$. The original formulation did not take this into account; it always walked over all lists $S_{\boldsymbol{C}}$ to separate them into the part that belongs to $NewB$ and the part that belongs to what remains in $\hat{B}$. With our terminology, it did:

```
for all constellations $\boldsymbol{C}$ such that $(\hat{B}, \boldsymbol{C})$ still needs postprocessing do
    for all new bottom states $s \in$ the original $\hat{B}$ with a transition to $\boldsymbol{C}$ do
        if $s \in NewB$ then
            Register that $(NewB, \boldsymbol{C})$ needs postprocessing.
            Move $s$ from $S_{\boldsymbol{C}}$ for $\hat{B}$ to the corresponding list for $NewB$.
            if $S_{\boldsymbol{C}}$ for $\hat{B}$ is empty then
                Register that $(\hat{B}, \boldsymbol{C})$ no longer needs postprocessing.
            end if
        end if
    end for
end for
```

If $NewB$ is much smaller than $\hat{B}$, a budget overrun may result because the loop still spends (a little) time for each state in $\hat{B} \setminus NewB$ with a transition to $\boldsymbol{C}$. This is illustrated in our second counterexample.

Assume that the partition in Figure 2 has been reached. Then, we choose SpB as splitter. the 🟠-state is a (weak) predecessor of SpB, but not of $\boldsymbol{SpC} \setminus SpB$, and therefore is split off from the remainder of $RfnB$. This turns all 🟦-states into new bottom states. First, it is registered that $(RfnB, \boldsymbol{E}_1)$, . . . , $(RfnB, \boldsymbol{E}_n)$ all need postprocessing. (Also $(RfnB, \boldsymbol{SpC})$ needs postprocessing, but we will disregard it in the lower bound for timing.) Then, one for one, these pairs are handled. Suppose it starts with $(RfnB, \boldsymbol{E}_1)$. The algorithm will find that it has to split $RfnB$ into two parts, namely the 🟦-state that is a predecessor of $\boldsymbol{E}_1$ and the $n-1$ other 🟦-states. Then, it walks over the $n-1$ pairs $(\mathit{RfnB}, \boldsymbol{E}_i)$ that still need postprocessing and their lists $S_{\boldsymbol{E}_i}$, containing altogether $2+3+\cdots+n$ states; from each list, it will remove the first 🟦-state. In total, $\frac{1}{2}n(n-1)-1$ list entries are read or removed. After that, the algorithm may handle $(\mathit{RfnB}, \boldsymbol{E}_2)$, split off one more 🟦-state from the rest, and walk over the $n-2$ remaining pairs $(\mathit{RfnB}, \boldsymbol{E}_i)$. Here, $\frac{1}{2}(n-1)(n-2)-1$ list entries are read or removed. For all the pairs up to $(\mathit{RfnB}, \boldsymbol{E}_n)$, the algorithm reads and finally removes $\Theta\left(n^3\right)$ list entries.

The Kripke structure in Figure 2 has $\mathcal{O}(n)$ states and $\mathcal{O}\left(n^2\right)$ transitions. Therefore, the algorithm should run in time $\mathcal{O}\left(n^2 \log n\right)$. However, it takes $\Omega\left(n^3\right)$ time. When we think of variants of this Kripke structure (e. g. reduced outdegree of the 🟠-state, or multiple 🟦-states with a transition to the same ◆-state), we find that there are actually d iterations over $\Theta(m)$ states.

Lemma 3. *Postprocessing new bottom states as described in [4] has a worst-case time complexity of* $\Omega(md)$.

3.4 A faster PostprocessNewBottom

The main idea for correcting the time bound was already hinted at earlier: Lines 3.10–3.14 try to distribute $S_{\boldsymbol{C}}$ over *NewB* and what remains of $\hat{B}$ in time proportional to the outgoing transitions of *NewB*, while keeping the order of $S_{\boldsymbol{C}}$ in line with the order of *NewBott*. This can be achieved if one distributes $S_{\boldsymbol{C}}$ simultaneously with distributing the states in $\hat{B}$ and their outgoing transitions themselves in Line 4.31. If all else fails, even constructing $S_{\boldsymbol{C}}$ for *NewB* from scratch can fit the time bound $\mathcal{O}(|\mathit{out}(\mathit{NewB})|)$.

3.5 PostprocessNewBottom is still too slow

The algorithm TrySplit′ walks through *all* new bottom states to look for blue states: in Line 3.9, we call it with $\mathit{MaybeBlue} = \mathit{NewBott}$. There may be a problem in a Kripke structure with a large number of new bottom states when we have to split over and over again for the same constellation, because every now and then, another new bottom state is discovered.

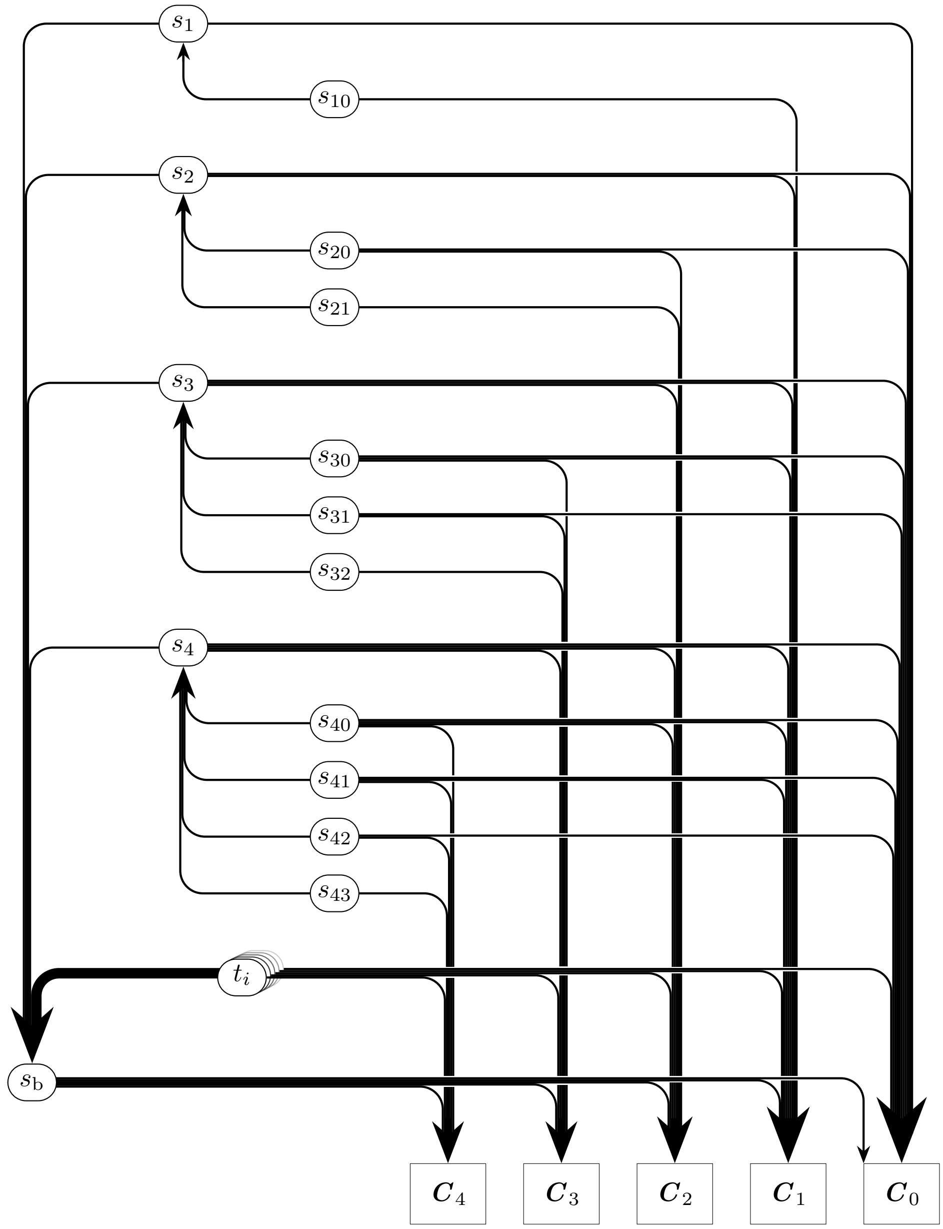
Consider the Kripke structure drawn partially in Figure 3. The states drawn individually are all in the same block, which is considered for refinement with $\mathit{SpB} \subset \boldsymbol{C}_0$ as splitter. We assume that state s_b has a transition to SpB, and all other transitions to $\boldsymbol{C}_0$ go to $\boldsymbol{C}_0 \setminus \mathit{SpB}$. Before splitting, s_b is the only bottom state in the block; however, as soon as it is split off, $s_1, s_2, \ldots$ become new bottom states.

If we extend this construction until we reach s_n and $s_{n,n-1}$, it will have $\mathcal{O}\left(n^2\right)$ states and $\mathcal{O}\left(n^3\right)$ transitions. For a better overview, we present the refinements in two levels; the algorithm follows the order of presentation, but does so iteratively.

- Let's start with $\boldsymbol{\mathit{SpC}} = \boldsymbol{C}_0$, after splitting it into *SpB* and the rest: this refinement splits s_b from the rest of the block and registers that $s_1, s_2, \ldots$ are new bottom states. As a result, the algorithm calls PostprocessNewBottom.
- Let's refine with $\boldsymbol{C}_1$: this will split s_1 from the block and create another new bottom state s_{10}. So we have to check for $\boldsymbol{C}_0$ again; this will now split off s_{10} from the rest.
- Then refine with $\boldsymbol{C}_2$: this splits s_2 from the block and creates two more new bottom states s_{20} and s_{21}. We now have to refine for $\boldsymbol{C}_0$ and $\boldsymbol{C}_1$ again: first s_{21} is split off and then s_{20}.
- Then we refine with $\boldsymbol{C}_3$: this will split s_3 from the block and create three more new bottom states s_{30}, s_{31} and s_{32}. Now we can refine for $\boldsymbol{C}_0$, $\boldsymbol{C}_1$ and $\boldsymbol{C}_2$ again; every time, we will split one of these new bottom states off.
- etc.

The reader will see that in all these refinements, the set of blue states is very small; however, it is never empty. We are allowed to spend time proportional to the number of these blue states and their transitions. In total, there will be about $\Theta\left(n^2\right)$ such refinements. This on itself is not yet

Figure 3: PostprocessNewBottom is still too slow.



problematic. However, we can without increasing the complexity class of the Kripke structure, add another $\mathcal{O}\left(n^2\right)$ new bottom states $t_1, \ldots, t_{n^2}$ to this block with transitions to each of the $\boldsymbol{C}_i$. Every time we look for the blue states, we have to run through all new bottom states to find the states that are not in $S_{\boldsymbol{C}_i}$. So, in reality, each refinement will spend time $\Omega\left(n^2\right)$. However, as there were only $\mathcal{O}\left(n^3\right)$ transitions, this is too much.

3.6 Possible correction

We should make sure that first those new bottom states are handled that have been found last: When we call TrySplit′ in Line 3.9, the blue coroutine first walks through the newest bottom states. Then, as soon as we know that all blue new bottom states have been found (based on the difference between the length of $S_{\boldsymbol{C}_i}$ and the total number of new bottom states), we can stop running through the other new bottom states. As a consequence, the new bottom states found earlier (and possibly already run through earlier) will not be visited another time; the time spent to skip over states in *MaybeBlue* \ *Blue* (in Line 4.12ℓ), over all refinements together, will be bounded by the number of outgoing transitions of new bottom states.

Another solution might be to handle all refinements w. r. t. a fixed set of new bottom states before considering more new bottom states. In terms of the original data structure, that would mean to keep the sets $X_B, X_{B'}$ untouched until all refinements are done, and only after that handle the new bottom states contained therein.

4 Conclusion

We showed that the algorithm of [4] for computing stuttering equivalence does not meet the acclaimed bound of $\mathcal{O}(m \log n)$; instead, there are examples showing it required $\Omega(md)$ time with d the maximum outdegree in the Kripke structure.

Presentation of the algorithm in pseudocode enabled us to identify the parts of the algorithm that are responsible for the overrun of the time bound. We are convinced that, after correcting TrySplit′ and PostprocessNewBottom, the time budgets (as indicated in the pseudocode) are met. Therefore, we are emboldened to confirm the main result of [4]:

Theorem 4. *It is possible to calculate the stuttering equivalence of a Kripke structure in $\mathcal{O}(m \log n)$ time and $\mathcal{O}(m)$ memory (by using the corrected Algorithm 1).*

References

[1] M. C. Browne, E. M. Clarke, and O. Grümberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical computer science*, 59(12):115–131, July 1988.

[2] Rob J. van Glabbeek and W. Peter Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, May 1996.

[3] Jan Friso Groote and Frits Vaandrager. An efficient algorithm for branching and stuttering equivalence. In M. S. Paterson, editor, *Automata, languages and programming*, volume 443 of *LNCS*, pages 626–638. Springer, Berlin, 1990.

[4] Jan Friso Groote and Anton Wijs. An $O(m \log n)$ algorithm for stuttering equivalence and branching bisimulation. arXiv e-prints 1601.01478, 2016. http://arxiv.org/abs/1601.01478.

[5] Jan Friso Groote and Anton Wijs. An $O(m \log n)$ algorithm for stuttering equivalence and branching bisimulation. In Marsha Chechik and Jean-François Raskin, editors, *Tools and algorithms for the construction and analysis of systems: TACAS*, volume 9636 of *LNCS*, pages 607–624. Springer, Berlin, 2016.