An $O(m \log n)$ Algorithm for Stuttering Equivalence and Branching Bisimulation

Jan Friso Groote(✉) and Anton Wijs

Department of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{J.F.Groote,A.J.Wijs}@tue.nl

Abstract. We provide a new algorithm to determine stuttering equivalence with time complexity $O(m \log n)$, where n is the number of states and m is the number of transitions of a Kripke structure. This algorithm can also be used to determine branching bisimulation in $O(m(\log |Act| + \log n))$ time. Theoretically, our algorithm substantially improves upon existing algorithms which all have time complexity $O(mn)$ [2,3,9]. Moreover, it has better or equal space complexity. Practical results confirm these findings showing that our algorithm can outperform existing algorithms with orders of magnitude, especially when the sizes of the Kripke structures are large.

1 Introduction

Stuttering equivalence [4] and branching bisimulation [8] were proposed as alternatives to Milner's weak bisimulation [13]. They are very close to weak bisimulation, with as essential difference that all states in the mimicking sequence $\tau^* a\, \tau^*$ must be related to either the state before or directly after the a from the first system. This means that branching bisimulation and stuttering equivalence are slightly stronger notions than weak bisimulation.

In [9] an $O(mn)$ time algorithm was proposed for stuttering equivalence and branching bisimulation, where m is the number of transitions and n is the number of states in either a Kripke structure (for stuttering equivalence) or a labelled transition system (for branching bisimulation). We refer to this algorithm as GV. It is based upon the $O(mn)$ algorithm for bisimulation equivalence in [11]. Both algorithms require $O(m+n)$ space. They calculate for each state whether it is bisimilar to another state.

The basic idea of the algorithms of [9,11] is to partition the set of states into blocks. States that are bisimilar always reside in the same block. Whenever there are some states in a block B' from which a transition is possible to some block B and there are other states in B' from which such a step is not possible, B' is split accordingly. Whenever no splitting is possible anymore, the partition is called stable, and two states are in the same block iff they are bisimilar.

There have been some attempts to come up with improvements of GV. The authors of [2] observed that GV only splits a block in two parts at a time.

© Springer-Verlag Berlin Heidelberg 2016
M. Chechik and J.-F. Raskin (Eds.): TACAS 2016, LNCS 9636, pp. 607–624, 2016.
DOI: 10.1007/978-3-662-49674-9_40

They proposed to split a block in as many parts as possible, reducing moving states and transitions to new blocks. Their worst case time and space complexities are worse than that of GV, especially the space complexity $O(mn)$, but in practice this algorithm can outperform GV. In [3], the space complexity is brought back to $O(m+n)$. A technique to be performed on Graphics Processing Units based on both GV and [2,3] is proposed in [19]. This improves the required runtime considerably by employing parallelism, but it does not imply any improvement to the single-threaded algorithm.

In [15] an $O(m \log n)$ algorithm is proposed for strong bisimulation as an improvement upon the algorithm of [11]. The core idea for this improvement is described as "process the smaller half" [1]. Whenever a block is split in two parts the amount of work must be contributed to the size of the smallest resulting block. In such a case a state is only involved in the process of splitting if it resides in a block at most half the size of the block it was previously in when involved in splitting. This means that a state can never be involved in more than $\log_2 n$ splittings. As the time used in each state is proportional to the number of incoming or outgoing transitions in that state, the total required time is $O(m \log n)$.

In this paper we propose the first algorithm for stuttering equivalence and branching bisimulation in which the "process the smaller half"-technique is used. By doing so, we can finally confirm the conjecture in [9] that such an improvement of GV is conceivable. Moreover, we achieve an even lower complexity, namely $O(m \log n)$, than conjectured in [9] by applying the technique twice, the second time for handling the presence of inert transitions. First we establish whether a block can be split by combining the approach regarding bottom states from GV with the detection approach in [15]. Subsequently, we use the "process the smaller half"-technique again to split a block by only traversing transitions in a time proportional to the size of the smallest subblock. As it is not known which of the two subblocks is smallest, the transitions of the two subblocks are processed alternatingly, such that the total processing time can be contributed to the smallest block. For checking behavioural equivalences, applying such a technique is entirely new. We are only aware of a similar approach for an algorithm in which the smallest bottom strongly connected component of a graph needs to be found [5].

Compared to checking other equivalences the existing algorithms for branching bisimulation/stuttering equivalence were already known to be practically very efficient. This is the reason that they are being used in multiple explicit-state model checkers, such as Cadp [7], the mCRL2 toolset [10] and TVT [18]. In particular they are being used as preprocessing steps for other equivalences (weak bisimulation, trace based equivalences) that are much harder to compute. For weak bisimulation recently an $O(mn)$ algorithm has been devised [12,16], but until that time an expensive transitive closure operation of at best $O(n^{2.373})$ was required. The improvements of our new algorithm are not restricted to stuttering equivalence and branching bisimulation alone, but they can also impact the computation time of all other behavioural equivalences.

Although our algorithm theoretically outperforms its predecessors substantially, we wanted to know whether it would also do so in practice. We find that

for dedicated examples our algorithm lives up to its theoretical improvement outperforming the existing algorithms in accordance with the theory. For practical examples we see that our algorithm can always match the best running times of existing algorithms, but especially when the Kripke structures and transition systems get large, our algorithm tends to outperform existing algorithms with orders of magnitude.

2 Preliminaries

We introduce Kripke structures and (divergence-blind) stuttering equivalence. In Sect. 6 we explain branching bisimulation and its application to labelled transition systems.

Definition 1. *A Kripke structure is a four tuple $K = (S, AP, \rightarrow, L)$, where*

1. *S is a finite set of states.*
2. *AP is a finite set of atomic propositions.*
3. *$\rightarrow \subseteq S \times S$ is a total transition relation, i.e., for each $s \in S$ there is an $s' \in S$ s.t. $s \rightarrow s'$.*
4. *$L : S \rightarrow 2^{AP}$ is a state labelling.*

We use $n{=}|S|$ for the number of states and $m{=}|\rightarrow|$ for the number of transitions. For a set of states $B{\subseteq}S$, we write $s \rightarrow_B s'$ for $s \rightarrow s'$ and $s' \in B$.

Definition 2. *Let $K = (S, AP, \rightarrow, L)$ be a Kripke structure. A symmetric relation $R \subseteq S \times S$ is a divergence-blind stuttering bisimulation iff for all $s, t \in S$ such that sRt:*

1. *$L(s) = L(t)$.*
2. *for all $s' \in S$ if $s \rightarrow s'$, then there are $t_0, \ldots, t_k \in S$ for some $k \in \mathbb{N}$ such that $t = t_0$, sRt_i, $t_i \rightarrow t_{i+1}$, and $s'Rt_k$ for all $i < k$.*

We say that two states $s, t \in S$ are divergence-blind stuttering equivalent, notation $s \leftrightarrows_{dbs} t$, iff there is a divergence-blind stuttering equivalence relation R such that sRt.

An important property of divergence-blind stuttering equivalence is that if states on a loop all have the same label then all these states are divergence-blind stuttering equivalent. We define stuttering equivalence in terms of divergence-blind stuttering equivalence using the following Kripke structure.

Definition 3. *Let $K = (S, AP, \rightarrow, L)$ be a Kripke structure. Define the Kripke structure $K_d = (S \cup \{s_d\}, AP \cup \{d\}, \rightarrow_d, L_d)$ where d is an atomic proposition not occurring in AP and s_d is a fresh state not occurring in S. Furthermore,*

1. *$\rightarrow_d \; = \; \rightarrow \cup \{\langle s, s_d \rangle \mid s$ is on a cycle of states all labelled with $L(s)$, or $s = s_d\}$.*
2. *For all $s \in S$ we define $L_d(s) = L(s)$ and $L_d(s_d) = \{d\}$.*

States $s, t \in S$ are stuttering equivalent, notation $s \leftrightarroweq_s t$ iff there is a divergence-blind stuttering bisimulation relation R on S_d such that sRt.

Note that an algorithm for divergence-blind stuttering equivalence can also be used to determine stuttering equivalence by employing only a linear time and space transformation. Therefore, we only concentrate on an algorithm for divergence-blind stuttering equivalence.

3 Partitions and Splitters: A Simple Algorithm

Our algorithms perform partition refinement of an initial partition containing the set of states S. A *partition* $\pi = \{B_i \subseteq S \mid 1 \leq i \leq k\}$ is a set of non empty subsets such that $B_i \cap B_j = \emptyset$ for all $1 \leq i < j \leq k$ and $S = \bigcup_{1 \leq i \leq k} B_i$. Each B_i is called a *block*.

We call a transition $s \rightarrow s'$ *inert w.r.t.* π iff s and s' are in the same block $B \in \pi$. We say that a partition π *coincides* with divergence-blind stuttering equivalence when $s \leftrightarroweq_{dbs} t$ iff there is a block $B \in \pi$ such that $s, t \in B$. We say that a partition *respects* divergence-blind stuttering equivalence iff for all $s, t \in S$ if $s \leftrightarroweq_{dbs} t$ then there is some block $B \in \pi$ such that $s, t \in B$. The goal of the algorithm is to calculate a partition that coincides with divergence-blind stuttering equivalence. This is done starting with the initial partition π_0 consisting of blocks B satisfying that if $s, t \in B$ then $L(s) = L(t)$. Note that this initial partition respects divergence-blind stuttering equivalence.

We say that a partition π is *cycle-free* iff for each block $B \in \pi$ there is no state $s \in B$ such that $s \rightarrow_B s_1 \ \rightarrow_B \cdots \rightarrow_B s_k \rightarrow s$ for some $k \in \mathbb{N}$. It is easy to make the initial partition π_0 cycle-free by merging all states on a cycle in each block into a single state. This preserves divergence-blind stuttering equivalence and can be performed in linear time employing a standard algorithm to find strongly connected components [1].

The initial partition is refined until it coincides with divergence-blind stuttering equivalence. Given a block B' of the current partition and the union $\boldsymbol{B}$ of some of the blocks in the partition, we define

$$\begin{array}{l} \mathit{split}(B', \boldsymbol{B}) = \{s_0 {\in} B' \mid \exists k {\in} \mathbb{N}, s_1, .., s_k {\in} S. s_i \rightarrow s_{i+1}, s_i {\in} B' \text{ for all } i < k \wedge s_k {\in} \boldsymbol{B}\} \\ \mathit{cosplit}(B', \boldsymbol{B}) = B' \setminus \mathit{split}(B', \boldsymbol{B}). \end{array}$$

Note that if $B' \subseteq \boldsymbol{B}$, then $\mathit{split}(B', \boldsymbol{B}) = B'$. It is common to split blocks under single blocks, i.e., $\boldsymbol{B}$ corresponding with a single block $B \in \pi$ [9,11]. However, as indicated in [15], it is required to split under the union of some of the blocks in π to obtain an $O(m \log n)$ algorithm. We refer to such groups of blocks as *constellations*. In Sect. 4, we use constellations consisting of more than one block when splitting.

We say that a block B' is *unstable* under $\boldsymbol{B}$ iff $\mathit{split}(B', \boldsymbol{B}) \neq \emptyset$ and $\mathit{cosplit}(B', \boldsymbol{B}) \neq \emptyset$. A partition π is *unstable* under $\boldsymbol{B}$ iff there is at least one $B' \in \pi$ which is unstable under $\boldsymbol{B}$. If π is not unstable under $\boldsymbol{B}$ then it is called *stable under* $\boldsymbol{B}$. If π is stable under all $\boldsymbol{B}$, then it is simply called stable.

A *refinement* of $B' \in \pi$ under $\boldsymbol{B}$ consists of two new blocks $\mathit{split}(B', \boldsymbol{B})$ and $\mathit{cosplit}(B', \boldsymbol{B})$. A partition π' is a refinement of π under $\boldsymbol{B}$ iff all unstable blocks $B' \in \pi$ have been replaced by new blocks $\mathit{split}(B', \boldsymbol{B})$ and $\mathit{cosplit}(B', \boldsymbol{B})$.

The following lemma expresses that if a partition is stable then it coincides with divergence-blind stuttering equivalence. It also says that during refinement, the encountered partitions respect divergence-blind stuttering equivalence and remain cycle-free.

Lemma 1. *Let $K = (S, AP, \rightarrow, L)$ be a Kripke structure and π a partition of S.*

1. *For all states $s, t \in S$, if $s, t \in B$ with B a block of the partition π, π is stable, and a refinement of the initial partition π_0, then $s \leftrightarrows_{dbs} t$.*
2. *If π respects divergence-blind stuttering equivalence then any refinement of π under the union of some of the blocks in π also respects it.*
3. *If π is a cycle-free partition, then any refinement of π is also cycle-free.*

Proof. 1. We show that if π is a stable partition, the relation $R = \{\langle s, t\rangle \mid s, t \in B,\ B \in \pi\}$ is a divergence-blind stuttering equivalence. It is clear that R is symmetric. Assume sRt. Obviously, $L(s) = L(t)$ because $s, t \in B$ and B refines the initial partition. For the second requirement of divergence-blind stuttering equivalence, suppose $s \rightarrow s'$. There is a block B' such that $s' \in B'$. As π is stable, it holds for t that $t = t_0 \rightarrow t_1 \rightarrow \cdots \rightarrow t_k$ for some $k \in \mathbb{N}$, $t_0, \ldots, t_{k-1} \in B$ and $t_k \in B'$. This clearly shows that for all $i < k$ sRt_i, and $s'Rt_k$. So, R is a divergence-blind stuttering equivalence, and therefore it holds for all states $s, t \in S$ that reside in the same block of π that $s \leftrightarrows_{dbs} t$.

2. The second part can be proven by reasoning towards a contradiction. Let us assume that a partition π' that is a refinement of π under $\boldsymbol{B}$ does not respect divergence-blind stuttering equivalence, although π does. Hence, there are states $s, t \in S$ with $s \leftrightarrows_{dbs} t$ and a block $B' \in \pi$ with $s, t \in B'$ and s and t are in different blocks in π'. Given that π' is a refinement of π under $\boldsymbol{B}$, $s \in \mathit{split}(B', \boldsymbol{B})$ and $t \in \mathit{cosplit}(B', \boldsymbol{B})$ (or vice versa, which can be proven similarly). By definition of *split*, there are $s_1, \ldots, s_{k-1} \in B'$ ($k \in \mathbb{N}$) and $s_k \in \boldsymbol{B}$ such that $s \rightarrow s_1 \rightarrow \cdots \rightarrow s_k$. Then, either $k = 0$ and $B' \subseteq \boldsymbol{B}$, but then $t \notin \mathit{cosplit}(B', \boldsymbol{B})$. Or $k > 0$, and since $s \leftrightarrows_{dbs} t$, there are $t_1, \ldots, t_{l-1} \in B'$ ($l \in \mathbb{N}$) and $t_l \in \boldsymbol{B}$ such that $t \rightarrow t_1 \rightarrow \cdots \rightarrow t_l$ with $s_i R t_j$ for all $1 \leq i < k$, $1 \leq j < l$ and $s_k R t_l$. This means that we have $t \in \mathit{split}(B', \boldsymbol{B})$, again contradicting that $t \in \mathit{cosplit}(B', \boldsymbol{B})$.

3. If π is cycle-free, this property is straightforward, since splitting any block of π will not introduce cycles. □

This suggests the following simple algorithm which has time complexity $O(mn)$ and space complexity $O(m+n)$, which was essentially presented in [9].

```
π := π0, i.e., the initial partition;
while π is unstable under some B ∈ π
    π := refinement of π under B;
```

It is an invariant of this algorithm that π respects divergence-blind stuttering equivalence and π is cycle-free. In particular, $\pi = \pi_0$ satisfies this invariant

initially. If π is not stable, a refinement under some block B exists, splitting at least one block. Therefore, this algorithm finishes in at most $n-1$ steps as during each iteration of the algorithm the number of blocks increases by one, and the number of blocks can never exceed the number of states. When the algorithm terminates, π is stable and therefore, two states are divergence-blind stuttering equivalent iff they are part of the same block in the final partition. This end result is independent of the order in which splitting took place.

In order to see that the time complexity of this algorithm is $O(mn)$, we must show that we can detect that π is unstable and carry out splitting in time $O(m)$. The crucial observation to efficiently determine whether a partition is stable stems from [9] where it was shown that it is enough to look at the bottom states of a block, which always exist for each block because the partition is cycle-free. The *bottom states* of a block are those states that do not have an outgoing inert transition, i.e., a transition to a state in the same block. They are defined by

$$\mathit{bottom}(B) = \{s \in B \mid \text{there is no state } s' \in B \text{ such that } s \rightarrow s'\}.$$

The following lemma presents the crucial observation concerning bottom states.

Lemma 2. *Let $K = (S, AP, \rightarrow, L)$ be a Kripke structure and π be a cycle-free partition of its states. Partition π is unstable under union $\boldsymbol{B}$ of some of the blocks in π iff there is a block $B' \in \pi$ such that*

$$\mathit{split}(B', \boldsymbol{B}) \neq \emptyset \text{ and } \mathit{bottom}(B') \cap \mathit{split}(B', \boldsymbol{B}) \subset \mathit{bottom}(B').$$

Here $\subset$ is meant to be a strict subset.

Proof. $\Rightarrow$ If π is unstable, then $\mathit{split}(B', \boldsymbol{B}) \neq \emptyset$ and $\mathit{split}(B', \boldsymbol{B}) \neq B'$. The first conjunct corresponds with the first condition. If $\mathit{split}(B', \boldsymbol{B}) \neq B'$, there are states $s \notin \mathit{split}(B', \boldsymbol{B})$. As the blocks $B' \in \pi$ do not have cycles, consider such an $s \notin \mathit{split}(B', \boldsymbol{B})$ with a smallest distance to a state $s_k \in \mathit{bottom}(B')$, i.e., $s \rightarrow s_1 \rightarrow \cdots \rightarrow s_k$ with all $s_i \in B'$. If s itself is an element of $\mathit{bottom}(B')$, the second part of the right hand side of the lemma follows. Assume $s \notin \mathit{bottom}(B')$, there is some state $s' \in B'$ closer to $\mathit{bottom}(B')$ such that $s \rightarrow s'$. Clearly, $s' \notin \mathit{split}(B', \boldsymbol{B})$ either, as otherwise $s \in \mathit{split}(B', \boldsymbol{B})$. But as s' is closer to $\mathit{bottom}(B')$, the state s was not a state with the smallest distance to a state in $\mathit{bottom}(B')$, which is a contradiction.

$\Leftarrow$ It follows from the right hand side that $\mathit{split}(B', \boldsymbol{B}) \neq \emptyset$, $\mathit{split}(B', \boldsymbol{B}) \neq B'$. □

This lemma can be used as follows to find a block to be split. Consider each $B \in \pi$. Traverse its incoming transitions and mark the states that can reach B in zero or one step. If a block B' has marked states, but not all of its bottom states are marked, the condition of the lemma applies, and it needs to be split. It is at most needed to traverse all transitions to carry this out, so its complexity is $O(m)$.

If B is equal to B', no splitting is possible. We implement it by marking all states in B as each state in B can reach itself in zero steps. In this case condition $\mathit{bottom}(B') \cap \mathit{split}(B', \boldsymbol{B}) \subset \mathit{bottom}(B')$ is not true. This is different from [9] where a block is never considered as a splitter of itself, but we require this in the algorithm in the next sections.

If a block B' is unstable, and all states from which a state in B can be reached in one step are marked, then a straightforward recursive procedure is required to extend the marking to all states in $\mathit{split}(B', B)$, and those states need to be moved to a new block. This takes time proportional to the number of transitions in B', i.e., $O(m)$.

4 Constellations: An $O(m \log n)$ Algorithm

The crucial idea to transform the algorithm from the previous section into an $O(m \log n)$ algorithm stems from [15]. By grouping the blocks in the current partition π into constellations such that π is stable under the union of the blocks in such a constellation, we can determine whether a block exists under which π is unstable by only looking at blocks that are at most half the size of the constellation, i.e., $|B| \leq \frac{1}{2}|\boldsymbol{B}|$, where $|\boldsymbol{B}| = \Sigma_{B' \in \boldsymbol{B}}|B'|$, for a block B in a constellation $\boldsymbol{B}$. If a block $B' \in \pi$ is unstable under B, then we use a remarkable technique consisting of two procedures running alternatingly to identify the smallest block resulting from the split. The whole operation runs in time proportional to the smallest block resulting from the split. We involve the blocks in $\boldsymbol{B} \setminus B$ in the splitting without explicitly analysing the states contained therein (for convenience, we write $\boldsymbol{B} \setminus B$ instead of $\boldsymbol{B} \setminus \{B\}$).

Working with constellations in this way ensures for each state that whenever it is involved in splitting, i.e., if it is part of a block that is used to split or that is being split, this block is half the size of the previous block in which the state resided when it was involved in splitting. That ensures that each state can at most be $\log_2(n)$ times involved in splitting. When involving a state, we only analyse its incoming and outgoing transitions, resulting in an algorithm with complexity $O(m \log n)$. Although we require quite a number of auxiliary data structures, these are either proportional to the number of states or to the number of transitions. So, the memory requirement is $O(m+n)$.

In the following, the set of constellations also forms a partition, which we denote by $\mathcal{C}$. A constellation is a set of one or more blocks from the current partition π. If a constellation contains only one block, it is called *trivial*. The current partition π is stable with respect to each constellation in $\mathcal{C}$.

If a constellation $\boldsymbol{B} \in \mathcal{C}$ contains more than one block, we select one block $B \in \boldsymbol{B}$ which is at most half the size of $\boldsymbol{B}$, and move it to a new trivial constellation $\boldsymbol{B'}$. We check whether the current partition is stable under B and $\boldsymbol{B} \setminus B$ according to Lemma 2 by traversing the incoming transitions of states in B and marking the encountered states that can reach B in zero or one step. For all blocks B' that are unstable according to Lemma 2, we calculate $\mathit{split}(B', B)$ and $\mathit{cosplit}(B', B)$, as indicated below.

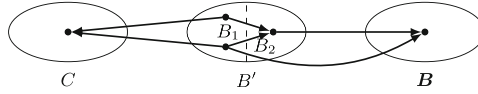

Fig. 1. After splitting B' under C, B_1 is not stable under $\boldsymbol{B}$.

As noted in [15], *cosplit*(B', B) is stable under $\boldsymbol{B} \setminus B$. Therefore, only further splitting of *split*(B', B) under $\boldsymbol{B} \setminus B$ must be investigated. If B' is stable under B because all bottom states of B' are marked, it can be that B' is not stable under $\boldsymbol{B} \setminus B$, which we do not address here explicitly, as it proceeds along the same line.

There is a special list data structure to recall for any B' and $\boldsymbol{B}$ which transitions go from B' to $\boldsymbol{B}$. When investigating whether *split*(B', B) is stable under B we adapt this list to determine the transitions from *split*(B', B) to $\boldsymbol{B} \setminus B$ and we simultaneously tag the states in B' that have a transition to $\boldsymbol{B} \setminus B$. Therefore, we know whether there are transitions from *split*(B', B) to $\boldsymbol{B} \setminus B$ and we can traverse the bottom states of *split*(B', B) to inspect whether there is a bottom state without a transition to B. Following Lemma 2, this allows us to determine whether *split*(B', B) must be split under $\boldsymbol{B} \setminus B$ in a time proportional to the size of B. How splitting is carried out is indicated below.

There is one aspect that complicates matters. If blocks are split, the new partition is not automatically stable under all constellations. This is contrary to the situation in [15] and was already observed in [9]. Figure 1 indicates the situation. Block B' is stable under constellation $\boldsymbol{B}$. But if B' is split under block C into B_1 and B_2, block B_1 is not stable under $\boldsymbol{B}$. The reason is, as exemplified by the following lemma, that some states that were non-bottom states in B' became bottom states in B_1.

Lemma 3. *Let $K = (S, AP, \rightarrow, L)$ be a Kripke structure with cycle free partition π with refinement π'. If π is stable under a constellation $\boldsymbol{B}$, and $B' \in \pi$ is refined into $B'_1, \ldots, B'_k \in \pi'$, then for each B'_i where the bottom states in B'_i are also bottom states in B', it holds that B'_i is also stable under $\boldsymbol{B}$.*

Proof. Assume B'_i is not stable under $\boldsymbol{B}$. This means that B'_i is not an element of $\boldsymbol{B}$. Hence, there is a state $s \in B'_i$ such that $s \rightarrow s'$ with $s' \in \boldsymbol{B}$ and there is a bottom state $t \in B'_i$ with no outgoing transition to a state in $\boldsymbol{B}$. But as B' was stable under $\boldsymbol{B}$, and s has an outgoing transition to a state in $\boldsymbol{B}$, all bottom states in B' must have at least one transition to a state in $\boldsymbol{B}$. Therefore, t cannot be a bottom state of B', and must have become a bottom state after splitting B'. $\square$

This means that if a block B' is the result of a refinement, and some of its states became bottom states, it must be made sure that B' is stable under the constellations. Typically, from the new bottom states a smaller number of blocks in the constellation can be reached. For each block we maintain a list of constellations that can be reached from states in this block. We match the

outgoing transitions of the new bottom states with this list, and if there is a block B'' reachable from states in the constellation, but not from the bottom states, B' must be split by B''.

The complexity of checking for additional splittings to regain stability when states become bottom states is only $O(m)$. Each state only becomes a bottom state once, and when that happens we perform calculations proportional to the number of outgoing transitions of this state to determine whether a split must be carried out.

It remains to show that splitting can be performed in a time proportional to the size of the smallest block resulting from the splitting. Consider splitting B' under $B \in \boldsymbol{B}$. While marking B' four lists of all marked and non marked, bottom and non bottom states have been constructed. We simultaneously mark states in B' either red or blue. Red means that there is a path from a state in B' to a state in B. Blue means that there is no such path. Initially, marked states are red, and non marked bottom states are blue.

This colouring is simultaneously extended to all states in B', spending equal time to both. The procedure is stopped when the colouring of one of the colours cannot be enlarged. We colour states red that can reach other red states via inert transitions using a simple recursive procedure. We colour states blue for which it is determined that all outgoing inert transitions go to a blue state (for this we need to recall for each state the number of outgoing inert transitions) and there is no direct transition to B. The marking procedure that terminates first, provided that its number of marked states does not exceed $\frac{1}{2}|B'|$, has the smallest block that must be split. Now that we know the smallest block we move its states to a newly created block.

Splitting regarding $\boldsymbol{B} \setminus B$ only has to be applied to $\mathit{split}(B', B)$, or, if all bottom states of B' were marked, to B'. As noted before $\mathit{cosplit}(B', B)$ is stable under $\boldsymbol{B} \setminus B$. Define $C := \mathit{split}(B', B)$ or $C := B'$ depending on the situation. We can traverse all bottom states of C and check whether they have outgoing transitions to $\boldsymbol{B} \setminus B$. This provides us with the blue states. The red states are obtained as we explicitly maintained the list of all transitions from C to $\boldsymbol{B} \setminus B$. By simultaneously extending this colouring the smallest subblock of either red or blue states is obtained and splitting can commence.

The algorithm is concisely presented in the box below. It is presented in full detail in Sect. 5 as the bookkeeping details of the algorithm are far from trivial.

$\pi :=$ initial partition; $\mathcal{C} := \{\pi\}$;
while $\mathcal{C}$ contains a non trivial constellation $\boldsymbol{B} \in \mathcal{C}$
  choose some $B \in \pi$ such that $B \in \boldsymbol{B}$ and $|B| \leq \frac{1}{2}|\boldsymbol{B}|$;
  $\mathcal{C}$:=partition $\mathcal{C}$ where $\boldsymbol{B}$ is replaced by B and $\boldsymbol{B} \setminus B$;
  if π is unstable for B or $\boldsymbol{B} \setminus B$
    $\pi' :=$ refinement of π under B and $\boldsymbol{B} \setminus B$;
    For each block $C \in \pi'$ with bottom states that were not bottom in π
      split C until it is stable for all constellations in $\mathcal{C}$;
    $\pi := \pi'$

5 Detailed Algorithm

This section presents the data structures and the algorithm in more detail.

5.1 Data Structures

As a basic data structure, we use (singly-linked) lists. For a list L of elements, we assume that for each element e, a reference to the position in L preceding the position of e is maintained, such that checking membership and removal can be done in constant time. In some cases we add extra information to the elements in the list. Moreover, for each list L, we maintain the size $|L|$ and pointers to its first and last element.

1. The current partition π consists of a list of blocks. Initially, it corresponds to π_0. All blocks are part of a single, initial constellation $\boldsymbol{C}_0$.
2. For each block B, we maintain the following:
   (a) A reference to the constellation containing B.
   (b) A list $B.btm\text{-}sts$ of the bottom states and a list $B.non\text{-}btm\text{-}sts$ of the other states.
   (c) A list $B.to\text{-}constlns$ of structures associated with constellations reachable via a transition from some $s{\in}B$. Initially, it contains one element associated with $\boldsymbol{C}_0$. Each element associated with some constellation $\boldsymbol{C}$ in this list also contains the following:
      – A reference *trans-list* to a list of all transitions from states in B to states in $\boldsymbol{C} \setminus B$ (note that transitions between states in B, i.e., inert transitions, are *not* in this list).
      – When splitting the block B into B and B' there is a reference in each list element to the corresponding list element in $B'.to\text{-}constlns$ (which in turn refers back to the element in $B.to\text{-}constlns$).
      – In order to check for stability when splitting produces new bottom states, each element contains a list to keep track of which new bottom states can reach the associated constellation.
   (d) A reference $B.inconstln\text{-}ref$ is used to refer to the element in $B.to\text{-}constlns$ associated with the constellation of B. It is used when a non-inert transition becomes inert and needs to be added to the *trans-list* of the element associated with that constellation.

   Furthermore, when splitting a block B' in constellation $\boldsymbol{B'}$ under a constellation $\boldsymbol{B}$ and block $B{\in}\boldsymbol{B}$, the following temporary structures are used, with $\boldsymbol{C}$ the new constellation to which B is moved:
   (a) A list $B'.mrkd\text{-}btm\text{-}sts$ contains marked states in B' with a transition to B.
   (b) A list $B'.mrkd\text{-}non\text{-}btm\text{-}sts$ contains states that are marked, but are not bottom states.
   (c) A reference $B'.constln\text{-}ref$ refers to the (new) element in $B'.to\text{-}constlns$ associated with constellation $\boldsymbol{C}$, i.e., the new constellation of B.

   (d) A reference B'.*coconstln-ref* is used to refer to the element in B'.*to-constlns* associated with constellation $\boldsymbol{B}$, i.e., the old constellation of B.
   (e) A list B'.*new-btm-sts* to keep track of the states that have become bottom states when B' was split. This is required to determine whether B' is stable under all constellations after a split.
3. Constellations are stored in two lists *trivial-constlns* and *non-trivial-constlns*. The first contains constellations consisting of exactly one block, while the latter contains the other constellations. Initially, if π_0 consists of one block, $\boldsymbol{C}_0$ is added to *trivial-constlns* and nothing needs to be done, because the initial partition is already stable. Otherwise $\boldsymbol{C}_0$ is added to *non-trivial-constlns*.
4. For each constellation, we maintain its list of blocks and its size (number of states).
5. Each transition $s \rightarrow s'$ refers with *to-constln-cnt* to the number of transitions from s to the constellation in which s' resides. For each state and constellation, there is one such variable, provided there is a transition from this state to this constellation.
   Each transition $s \rightarrow s'$ has a reference to the element associated with $\boldsymbol{B}$ in the list B.*to-constlns* where $s{\in}B$ and $s'{\in}\boldsymbol{B}$. This is denoted as $(s \rightarrow s')$.*to-constln-ref*. Initially, it refers to the single element in B.*to-constlns*, unless the transition is inert, i.e., both $s{\in}B$ and $s'{\in}B$.
   Furthermore, each transition $s \rightarrow s'$ is stored in the list of transitions from B to $\boldsymbol{B}$. Initially, there is such a list for each block in the initial partition π_0. From a transition $s \rightarrow s'$, the list can be accessed via $(s \rightarrow s')$.*to-constln-ref.trans-list*.
6. For each state $s{\in}B$ we maintain the following information:
   (a) A reference to the block containing s.
   (b) A static list $s.T_{tgt}$ of transitions of the form $s \rightarrow s'$ containing precisely all the transitions from s.
   (c) A static list $s.T_{src}$ of transitions $s' \rightarrow s$ containing all the transitions to s. We write such transitions as $s \leftarrow s'$, to stress that these move into s.
   (d) A counter s.*inert-cnt* containing the number of outgoing transitions to a state in the same block as s. For any bottom state s, we have $s.\mathit{inert\text{-}cnt} = 0$.
   (e) Furthermore, when splitting a block B' under $\boldsymbol{B}$ and $B{\in}\boldsymbol{B}$, there are references s.*constln-cnt* and s.*coconstln-cnt* to the variables that are used to count how many transitions there are from s to B and from s to $\boldsymbol{B} \setminus B$.

Figure 2 illustrates some of the used structures. A block B_1 in constellation $\boldsymbol{B}$ contains bottom states s_1, s_2' and non-bottom state s_2. For s_1, we have transitions $s_1 \rightarrow s_1'$, $s_1 \rightarrow s_1''$ to constellation $\boldsymbol{C}$. Both have the following references:
   (a) *to-constln-cnt* to the number of outgoing transitions from s_1 to $\boldsymbol{C}$.
   (b) *to-constln-ref* to the element $(\boldsymbol{C}, \bullet, \bullet)$ in B_1.*to-constlns*, where the $\bullet$'s are the (now uninitialized) references that are used when splitting.
   (c) Via $(\boldsymbol{C}, \bullet, \bullet)$, a reference *trans-list* to the list of transitions from B_1 to $\boldsymbol{C}$.

Note that for the inert transition $s_2 \rightarrow s_2'$, we only have a reference to the number of outgoing transitions from s_2 to $\boldsymbol{B}$.

Fig. 2. An example showing some of the data structures used in the detailed algorithm.

5.2 Finding the Blocks that Must Be Split

While *non-trivial-constlns* is not empty, we perform the algorithm listed in the following sections. To determine whether the current partition π is unstable, we select a constellation $\boldsymbol{B}$ in *non-trivial-constlns*, and we select a block B from $\boldsymbol{B}$ such that $|B| \leq \frac{1}{2}|\boldsymbol{B}|$. We first check which blocks are unstable for B and $\boldsymbol{B} \setminus B$.

1. Move B to a new trivial constellation $\boldsymbol{C}$. If $|\boldsymbol{B}.blocks|=1$, make $\boldsymbol{B}$ trivial.
2. For each state $s \in B$, do the steps below for each $s' \in B'$ such that $s \leftarrow s' \in s.T_{src}$, and $B \neq B'$.
   (a) If B' has no marked states, put it in a list *splittable-blks*, let $B'.coconstln\text{-}ref$ refer to $(s \leftarrow s').to\text{-}constln\text{-}ref$, $B'.constln\text{-}ref$ to a new element in $B'.to\text{-}constlns$.
   (b) Mark s'.
   (c) Let $s'.constln\text{-}cnt$ be the number of transitions to B and $s'.coconstln\text{-}cnt$ the number of remaining outgoing transitions. All outgoing transitions of s' must refer to the appropriate counter.
   (d) Move all visited transitions to $B'.constln\text{-}ref.trans\text{-}list$.
3. Next, check whether B itself can be split. Mark all states, add B to *splittable-blks* and reset $B.constln\text{-}ref$ and $B.coconstln\text{-}ref$. For each state $s \in B$, do the steps below for each $s' \in B' \in \boldsymbol{B'}$ such that $s \leftarrow s' \in s.T_{src}$, and either $\boldsymbol{B'}=\boldsymbol{B}$ or $\boldsymbol{B'}=\boldsymbol{C}$.
   (a) If $\boldsymbol{B'}=\boldsymbol{B}$, let $B.coconstln\text{-}ref$ refer to $(s \rightarrow s').to\text{-}constln\text{-}ref$ and $B.constln\text{-}ref$ and $B.inconstln\text{-}ref$ to a new element for $\boldsymbol{C}$ in $B.to\text{-}constlns$.
   (b) Update $s.constln\text{-}cnt$ and $s.coconstln\text{-}cnt$ as in step 2(c).
4. For each $B' \in splittable\text{-}blks$, if all its bottom states are marked and either there is no marked bottom state s with $s.coconstln\text{-}cnt=0$ or $B'.coconstln\text{-}ref.trans\text{-}list$ is empty, remove B' from *splittable-blks* and remove its temporary markings, i.e. unmark all states, reset the counters and references.
5. If *splittable-blks* is not empty, start splitting (Sect. 5.3). Else, select another non-trivial constellation $\boldsymbol{B}$ and block $B \in \boldsymbol{B}$, and continuing with step 1. If there are no non-trivial constellations left, the algorithm terminates.

5.3 Splitting the Blocks

Splitting the splittable blocks is performed using the following steps, in which the procedures used to simultaneously mark states when splitting a block are crucial for the performance. We refer to the whole operation as the *lockstep search* and call the two procedures **detect1** and **detect2**. In the lockstep search, these procedures alternatingly process a transition. The entire operation terminates when one of the procedures terminates. If one procedure acquires more than half the number of states in the block it works on, it is stopped and the other is allowed to terminate. We present **detect1** and **detect2** below; both get a list of states, D_1 and D_2, respectively, and a block K to work on as input. In addition, **detect2** takes a Boolean parameter indicating whether the splitting is a nested one, i.e., whether it directly follows an earlier split of the same block.

detect1($\boldsymbol{D_1}, \boldsymbol{K}$):

- Create empty stack Q, list L;
- While $|L| \leq \frac{1}{2}|K|$ and either $Q \neq \emptyset$ or end of D_1 not reached:
  - If $Q = \emptyset$ add next $s \in D_1$ to Q and L;
  - Pop s from Q. For all $s \leftarrow s' \in s.T_{src}$, if $s' \in K \wedge s' \notin L$, add s' to Q and L.

detect2($\boldsymbol{D_2}, \boldsymbol{K}, \boldsymbol{nested}$):

- Create empty priority queue P, list L';
- While $|L'| \leq \frac{1}{2}|K|$ and either P has prio. 0 states or end of D_2 not reached:
  - Take a state s from D_2 or with prio. 0 from P and add it to L';
  - For all $s \leftarrow s' \in s.T_{src}$, if $s' \in K \setminus (P \cup L')$, and $s' \notin$ *mrkd-non-btm-sts* or if *nested*, s' does not have a transition to $\boldsymbol{B} \setminus B$, add s' with prio. s'.*inert-cnt* to P;
  - If $s' \in P$, decrement priority of s'.

We walk through the blocks $B' \in \boldsymbol{B'}$ in *splittable-blks*, which must be split into two or three blocks under constellation $\boldsymbol{B}$ and block B. If all bottom states are marked, then we have $split(B', B) = B'$, and can start with step 3 below.

1. Launch a lockstep search with D_1 the list of marked states in B', D_2 the list B'.*btm-sts*, $K = B'$, and *nested* = **false**.
2. Depending on whether **detect1** or **detect2** terminated in the previous step, one of the lists L or L' contains the states to be moved to a new block B''. Below we refer to this list as N. For each $s \in N$, move s to B'', and do the following:
   (a) For each $s \to s' \in T_{tgt}$, do the following steps.
      i. If $(s \to s')$.*to-constln-ref* is initialized, check whether it refers to an new element in B''.*to-constlns*. If not, create it. If appropriate, set references B''.*inconstln-ref*, B''.*constln-ref* and B''.*coconstln-ref*. Move $s \to s'$ to the *trans-list* of the new element. If the related element in B'.*to-constlns* no longer holds transitions, remove it.

   ii. Else, if $s' \in B' \setminus N$ (a transition becomes non-inert), decrement $s.\mathit{inert\text{-}cnt}$. If $s.\mathit{inert\text{-}cnt}{=}0$, make s bottom, add $s \rightarrow s'$ to $B''.\mathit{inconstln\text{-}ref.trans\text{-}list}$ (if $B''.\mathit{inconstln\text{-}ref}$ does not exist, create it first).

   (b) For each $s \leftarrow s' \in T_{src}$, $s' \in B' \setminus N$ (an inert transition becomes non-inert), perform steps similar to 2(a).ii.

3. Next, we split $\mathit{split}(B', B)$ under $\boldsymbol{B} \setminus B$. Define $C{=}\mathit{split}(B', B)$. C is stable under $\boldsymbol{B} \setminus B$ if $C.\mathit{coconstln\text{-}ref}$ is uninitialized or holds an empty *trans-list*, or for all $s \in C.\mathit{mrkd\text{-}btm\text{-}sts}$ it holds that $s.\mathit{coconstln\text{-}cnt} > 0$. If this is not the case, then we launch a lockstep search with D_1 the list of states s occurring in some $s \rightarrow s'$ in $\mathit{split}(B', B).\mathit{coconstln\text{-}ref.trans\text{-}list}$, D_2 the list of states s with $s.\mathit{coconstln\text{-}cnt} = 0$ in $C.\mathit{mrkd\text{-}btm\text{-}sts}$, $K = C$, and $\mathit{nested} = \mathbf{true}$. Finally, we split C by moving the states in either L or L' to a new block B''', depending on which list is the smallest.
4. Remove the temporary markings of each block C resulting from the splitting of B'.
5. If the splitting of B' resulted in new bottom states, check for those states whether further splitting is required, i.e., whether from some of them, not all constellations can be reached which can be reached from the block. For all $\hat{B} \in \{B', B'', B'''\}$, new bottom states s, $s \rightarrow s' \in s.T_{tgt}$, add s to the states list of the element associated with $\bar{\boldsymbol{B}}$ in $\hat{B}.\mathit{to\text{-}constlns}$, where $s' \in \bar{\boldsymbol{B}}$, and move the element to the front of the list.
6. Perform the following steps for each block $\hat{B}$ with new bottom states, as long as there are such blocks.
   (a) Walk through the elements in $\hat{B}.\mathit{to\text{-}constlns}$. If the states list of an element associated with a constellation $\boldsymbol{B}$ does not contain all new bottom states, further splitting is required under $\boldsymbol{B}$:
      i. Launch a lockstep search with D_1 the list of states s occurring in some $s \rightarrow s'$ with $s' \in \boldsymbol{B}$ in the list *trans-list* associated with $\boldsymbol{B} \in \hat{B}.\mathit{to\text{-}constlns}$, D_2 the list of states $s \in \hat{B}.\mathit{new\text{-}btm\text{-}sts}$ minus the new bottom states that can reach $\boldsymbol{B}$, $K = \hat{B}$, and $\mathit{nested} = \mathbf{true}$.
      ii. Split $\hat{B}$ by performing step 2 to produce a new block $\hat{B}'$. Move all states in $\hat{B}.\mathit{new\text{-}btm\text{-}sts}$ that have moved to $\hat{B}'$ to $\hat{B}'.\mathit{new\text{-}btm\text{-}sts}$, and also move them from the states lists in the elements of $\hat{B}.\mathit{to\text{-}constlns}$ to the corresponding elements of $\hat{B}'.\mathit{to\text{-}constlns}$ (those elements refer to each other). If a states list becomes empty, move that element to the back of its list.
      iii. Perform step 5 for $\hat{B}$ and $\hat{B}'$.

   (b) If no further splitting was required for $\hat{B}$, empty $\hat{B}.\mathit{new\text{-}btm\text{-}sts}$ and clear the remaining states lists in $\hat{B}.\mathit{to\text{-}constlns}$.
7. If $\boldsymbol{B'} \in \mathit{trivial\text{-}constlns}$, move it to *non-trivial-constlns*.

6 Application to Branching Bisimulation

We show that the algorithm can also be used to determine branching bisimulation, using the transformation from [14,17], with complexity $O(m(\log|Act| +$

$\log n$)). Branching bisimulation is typically applied to labelled transition systems (LTSs). An LTS is a three tuple $A = (S, Act, \rightarrow)$, with S a finite set of states, Act a finite set of actions including the internal action τ, and $\rightarrow \subseteq S \times Act \times S$ a transition relation.

Definition 4. *Consider the LTS $A = (S, Act, \rightarrow)$. We call a symmetric relation $R \subseteq S \times S$ a* branching bisimulation relation *iff*

$$\forall s, t, s' \in S. \forall a \in Act. sRt \wedge s \xrightarrow{a} s' \implies$$
$$(a = \tau \wedge s'Rt) \vee (\exists t', t'' \in S. t \twoheadrightarrow t' \xrightarrow{a} t'' \wedge sRt' \wedge s'Rt''),$$

where $\twoheadrightarrow$ is the transitive, reflexive closure of $\xrightarrow{\tau}$.

States are *branching bisimilar* iff there is a branching bisimulation relation R relating them.

Our new algorithm can be applied to an LTS by translating it to a Kripke structure.

Definition 5. *Let $A = (S, Act, \rightarrow)$ be an LTS. We construct the embedding of A to be the Kripke structure $K_A = (S_A, AP, \rightarrow, L)$ as follows:*

1. *$S_A = S \cup \{\langle a, t\rangle \mid s \xrightarrow{a} t$ for some $t \in S\}$.*
2. *$AP = Act \cup \{\bot\}$.*
3. *$\rightarrow$ is the least relation satisfying ($s, t \in S$, $a \in Act \setminus \tau$): $\frac{s \xrightarrow{a} t}{s \longrightarrow \langle a,t\rangle}$, $\frac{}{\langle a,t\rangle \longrightarrow t}$ and $\frac{s \xrightarrow{\tau} t}{s \longrightarrow t}$.*
4. *$L(s) = \{\bot\}$ for $s \in S$ and $L(\langle a, t\rangle) = \{a\}$.*

The following theorem stems from [14].

Theorem 1. *Let A be an LTS and K_A its embedding. Then two states are branching bisimilar in A iff they are divergence-blind stuttering equivalent in K_A.*

If we start out with an LTS with n states and m transitions then its embedding has at most $m + n$ states and $2m$ transitions. Hence, the algorithm

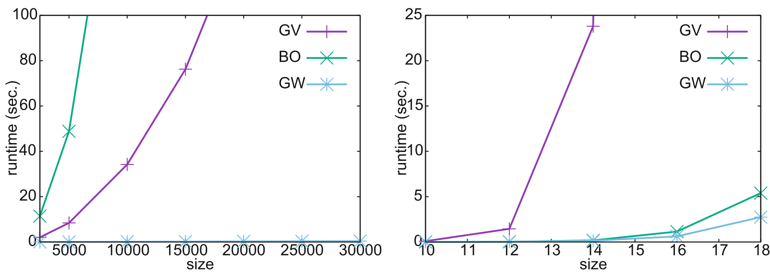

Fig. 3. Runtime results for $(a{\cdot}\tau)^{size}$ sequences (left) and trees of depth *size* (right)

Table 1. Runtime (in sec.) and memory use (in MB) results for GV, BO, and GW

Model	*n*	*m*	min. *n*	min. *m*	time GV	me. GV	time BO	me. BO	time GW	me. GW
vasy_40	40,006	60,007	20,003	40,004	142.77	65	762.69	62	**0.34**	93
vasy_65	65,537	2,621,480	65,536	2,621,440	239.67	437	47.88	645	**20.07**	2,481
vasy_66	66,929	1,302,664	51,128	1,018,692	**7.42**	208	16.16	356	9.05	853
vasy_69	69,754	520,633	69,753	520,632	**3.98**	155	12.65	171	4.53	493
vasy_116	116,456	368,569	22,398	87,674	3.84	95	15.73	128	**2.68**	142
vasy_157	157,604	297,000	3,038	12,095	6.98	97	6.80	110	**1.08**	129
vasy_164	164,865	1,619,204	992	3,456	**3.89**	251	20.20	316	5.38	246
vasy_166	166,464	651,168	42,195	197,200	21.60	153	6.20	177	**3.89**	376
cwi_214	214,202	684,419	478	1,612	**0.87**	140	29.92	197	2.64	140
cwi_371	371,804	641,565	2,134	5,634	42.70	179	17.37	261	**3.12**	168
cwi_566	566,640	3,984,157	198	791	1683.28	454	26.24	531	**19.94**	454
vasy_574	574,057	13,561,040	3,577	16,168	105.10	1,766	487.01	2,192	**40.18**	1,495
cwi_2165	2,165,446	8,723,465	4,256	20,880	80.56	1,403	387.93	2,409	**59.49**	1,948
cwi_2416	2,416,632	17,605,592	730	2,899	1,679.55	1,932	**59.29**	2,660	90.69	1,932
vasy_2581	2,581,374	11,442,382	704,737	3,972,600	2,592.74	1,690	463.52	2,344	**76.16**	5,098
vasy_4220	4,220,790	13,944,372	1,186,266	6,863,329	3,643.08	2,054	863.74	2,951	**119.20**	7,287
vasy_4338	4,338,672	15,666,588	704,737	3,972,600	5,290.54	2,258	587.87	3,026	**109.21**	6,927
vasy_6020	6,020,550	19,353,474	256	510	130.76	2,045	95.76	3,482	**45.54**	2,045
vasy_6120	6,120,718	11,031,292	2,505	5,358	546.11	1,893	291.30	2,300	**81.05**	3,392
cwi_7838	7,838,608	59,101,007	62,031	470,230	745.33	6,319	11,667.98	11,027	**617.46**	14,456
vasy_8082	8,082,905	42,933,110	290	680	288.45	6,098	677.28	7,824	**200.72**	6,108
vasy_11026	11,026,932	24,660,513	775,618	2,454,834	5,005.61	3,642	2,555.30	5,235	**225.20**	10,394
vasy_12323	12,323,703	27,667,803	876,944	2,780,022	5,997.26	4,068	2,068.52	5,770	**256.70**	11,575
cwi_33949	33,949,609	165,318,222	12,463	71,466	1,684.56	21,951	11,635.09	42,162	**1,459.92**	37,437
dining_14	18,378,370	164,329,284	228,486	2,067,856	1,264.67	20,155	3,010.17	31,201	**1,100.91**	20,155
1394-fin3	126,713,623	276,426,688	160,258	538,936	229,217.0	26,000	15,319.00	75,000	**1,516.00**	45,000

requires $O(m \log(n+m))$ time. As m is at most $|Act|n^2$ this is also equal to $O(m(\log|Act| + \log n))$.

As a final note, the algorithm can also be adapted to determine divergence-sensitive branching bisimulation [8], by adding a τ-self loop to those states on a τ-loop.

7 Experiments

The new algorithm has been implemented as part of the mCRL2 toolset [6], which offers implementations of GV and the algorithm by Blom and Orzan [2] that distinguishes states by their connection to blocks via their outgoing transitions. We refer to the latter as BO. The performance of GV and BO can be very different on concrete examples. We have extensively tested the new algorithm by applying it to thousands of randomly generated LTSs and comparing the results with those of the other algorithms.

We experimentally compared the performance of GV, BO, and the implementation of the new algorithm (GW). All experiments involve the analysis of LTSs, which for GW are first transformed to Kripke structures using the translation of Sect. 6. The reported runtimes do not include the time to read the input LTS and write the output, but the time it takes to translate the LTS to a Kripke structure and to reduce strongly connected components is included.

Practically all experiments have been performed on machines running CentOS Linux, with an Intel E5-2620 2.0 GHz CPU and 64 GB RAM. Exceptions

to this are the final two entries in Table 1, which were obtained by using a machine running FEDORA 12, with an INTEL XEON E5520 2.27 GHz CPU and 1 TB RAM.

Figure 3 presents the runtime results for two sets of experiments to demonstrate that GW has the expected scalability. At the left are the results of analysing single sequences of the shape $(a{\cdot}\tau)^n$. As the length $2n$ of such a sequence is increased, the results show that the runtimes of both BO and GV increase at least quadratically, while the runtime of GW grows linearly. All algorithms require n iterations, in which BO and GV walk over all the states in the sequence, but GW only moves two states into a new block. At the right of Fig. 3, the results are displayed of analysing trees of depth n that up to level $n-1$ correspond with a binary tree of τ-transitions. Each state at level $n-1$ has a uniquely labelled outgoing transition to a state in level n. BO only needs one iteration to obtain the stable partition. Still GW beats BO by repeatedly splitting off small blocks of size $2(k-1)$ if a state at level k is the splitter.

Table 1 contains results for minimising LTSs from the VLTS benchmark set[1] and the mCRL2 toolset[2]. These experiments demonstrate that also when applied to actual state spaces of real models, GW generally outperforms the best of the other algorithms, often with a factor 10 and sometimes with a factor 100. This difference tends to grow as the LTSs get larger. GW's memory usage is only sometimes substantially higher than GV's and BO's, which surprised us given the amount of required bookkeeping.

References

1. Aho, A., Hopcroft, J., Ullman, J.: The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading (1974)
2. Blom, S.C., Orzan, S.: Distributed branching bisimulation reduction of state spaces. In: FMICS 2003. ENTCS, vol. 80, pp. 109–123. Elsevier (2003)
3. Blom, S.C., van de Pol, J.C.: Distributed branching bisimulation minimization by inductive signatures. In: PDMC 2009. EPTCS, vol. 14, pp. 32–46. Open Publ. Association (2009)
4. Browne, M.C., Clarke, E.M., Grumberg, O.: Characterizing finite Kripke structures in propositional temporal logic. Theoret. Comput. Sci. **59**(1,2), 115–131 (1988)
5. Chatterjee, K., Henzinger, M.: Faster and dynamic algorithms for maximal end-component decomposition and related graph problems in probabilistic verification. In: SODA 2011, pp. 1318–1336. SIAM (2011)
6. Cranen, S., Groote, J.F., Keiren, J.J.A., Stappers, F.P.M., de Vink, E.P., Wesselink, W., Willemse, T.A.C.: An overview of the mCRL2 toolset and its recent advances. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 199–213. Springer, Heidelberg (2013)
7. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: a toolbox for the construction and analysis of distributed processes. Softw. Tools Technol. Transfer. **15**(2), 98–107 (2013)
8. van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. J. ACM **43**(3), 555–600 (1996)

[1] http://cadp.inria.fr/resources/vlts.
[2] http://www.mcrl2.org.

9. Groote, J.F., Vaandrager, F.W.: An efficient algorithm for branching bisimulation and stuttering equivalence. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 626–638. Springer, Heidelberg (1990)
10. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. The MIT Press, Cambridge (2014)
11. Kannelakis, P., Smolka, S.: CCS Expressions, Finite State Processes and Three Problems of Equivalence. Inf. Comput. **86**, 43–68 (1990)
12. Li, W.: Algorithms for computing weak bisimulation equivalence. In: TASE 2009, pp. 241–248. IEEE (2009)
13. Milner, R. (ed.): Calculus of Communicating Systems. Lecture Notes in Computer Science, vol. 92. Springer, Heidelberg (1980)
14. De Nicola, R., Vaandrager, F.W.: Three logics for branching bisimulation. Journal of the ACM **42**, 458–487 (1995)
15. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. **16**(6), 973–989 (1987)
16. Ranzato, F., Tapparo, F.: Generalizing the Paige-Tarjan algorithm by abstract interpretation. Inf. Comput. **206**(5), 620–651 (2008)
17. Reniers, M.A., Schoren, R., Willemse, T.A.C.: Results on embeddings between state-based and event-based systems. Comput. J. **57**(1), 73–92 (2014)
18. Virtanen, H., Hansen, H., Valmari, A., Nieminen, J., Erkkilä, T.: Tampere verification tool. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 153–157. Springer, Heidelberg (2004)
19. Wijs, A.: GPU accelerated strong and branching bisimilarity checking. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 368–383. Springer, Heidelberg (2015)