

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 where Act is the set of actions in a labelled transition system.

Theoretically, our algorithm substantially improves upon existing algorithms which all have time complexity $O(mn)$ at best [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.

The importance of our algorithm stretches far beyond stuttering equivalence and branching bisimulation. The known $O(mn)$ algorithms were already far more efficient (both in space and time) than most other algorithms to determine behavioural equivalences (including weak bisimulation) and therefore it was often used as an essential preprocessing step. This new algorithm makes this use of stuttering equivalence and branching bisimulation even more attractive.

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. 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 alternately, 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].

The algorithm that we propose is complex. Although the basic sketch of the algorithm is relatively straightforward, it heavily relies on auxiliary data structures. For instance, for each transition it is recalled how many other transitions there are from the block where the transition starts to the constellation in which the transition ends. Maintaining such auxiliary data structures is a tedious process. Therefore, we do not only prove the major steps of the algorithm correct, we also provide a very detailed description of the algorithm, and we ran the implemented algorithm on many thousands of randomly generated test cases, comparing the reductions with the outcomes of existing algorithms. This not only convinced us that the algorithm is correct and correctly implemented, it also allows others to easily reimplement the algorithm.

From a theoretical viewpoint our algorithm outperforms its predecessors substantially. But a fair question is whether this also translates into practice. The theoretical bounds are complexity upperbounds, and depending on the transition system, the classical algorithms can be much faster than the upperbound suggests. Furthermore, the increased bookkeeping in the new algorithm may be such a burden that all gains are lost. For this reason we compared the practical performance of our algorithm with that of the predecessors, and we found that for practical examples our algorithm can always match the best running times, but especially when the Kripke structures and transition systems get large, our algorithm tends to outperform existing algorithms with orders of magnitude.

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 a $O(mn)$ algorithm has been devised [12, 16], but until that time an expensive transitive closure operation of at best $O(n^{2.373})$ was required. Using our algorithm as a preprocessing step, the computation time of all other behavioural ‘weak’ equivalences can be made faster.

2 Preliminaries

We introduce Kripke structures and (divergence-blind) stuttering equivalence. Labelled transition systems and branching bisimulation will be addressed in section 6.

Definition 2.1 (Kripke structure). 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 $s' \in B$, and $s \rightarrow B$ iff there is some $s' \in B$ such that $s \rightarrow s'$. We write $s \not\rightarrow s'$ and $s \not\rightarrow B$ iff it is not the case that $s \rightarrow s'$, resp., $s \rightarrow B$.

Definition 2.2 (Divergence-blind stuttering equivalence). Let $K = (S, AP, \rightarrow, L)$ be a Kripke structure. A symmetric relation $R \subseteq S \times S$ is a *divergence-blind stuttering equivalence* 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 $t_0, \dots, t_k \in S$ 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 $s \stackrel{\leftrightarrow}{\text{abs}} t$, 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 2.3 (Stuttering equivalence). 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. $\rightarrow_d = \rightarrow \cup \{(s, s_d) \mid s \text{ is on a cycle of states all labelled with } L(s), \text{ or } s = s_d\}$.
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 $s \stackrel{\leftrightarrow}{s} t$ iff there is a divergence-blind stuttering equivalence 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 $s \stackrel{\leftrightarrow}{\text{abs}} t$ 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 $s \stackrel{\leftrightarrow}{\text{abs}} t$ 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 there is no state $s \in B$ such that $s \rightarrow_B s_1 \rightarrow_B \dots \rightarrow_B s_k \rightarrow s$ for some $k \in \mathbb{N}$ for each block $B \in \pi$. 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 \mathbf{B} of some of the blocks in the partition, we define

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

Note that if $B' \subseteq \mathbf{B}$, then $\text{split}(B', \mathbf{B}) = B'$. The sets $\text{split}(B', \mathbf{B})$ and $\text{cosplit}(B', \mathbf{B})$ are intended as the new blocks to replace B' . It is common to split blocks under single blocks, i.e., \mathbf{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 unions as *constellations*. In section 4, we use constellations consisting of more than one block in the splitting.

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

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

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 3.1. 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 $s \stackrel{\leftrightarrow}{\text{dbs}} t$.
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 $t = t_0 \rightarrow t_1 \rightarrow \dots \rightarrow t_k$ for some $k \in \mathbb{N}$, $t_0, \dots, t_{k-1} \in B$ 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 $s \stackrel{\leftrightarrow}{\text{dbs}} t$.
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 \mathbf{B} does not respect divergence-blind stuttering equivalence, although π does. Hence, there are states $s, t \in S$ with $s \stackrel{\leftrightarrow}{\text{dbs}} t$ 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 \mathbf{B} , $s \in \text{split}(B', \mathbf{B})$ and $t \in \text{cosplit}(B', \mathbf{B})$ (or vice versa, which can be proven similarly). By definition of *split*, there are $s_1, \dots, s_{k-1} \in B'$ ($k \in \mathbb{N}$) and $s_k \in \mathbf{B}$ such that $s \rightarrow s_1 \rightarrow \dots \rightarrow s_k$. Then, either $k = 0$ and $B' \subseteq \mathbf{B}$, but then $t \notin \text{cosplit}(B', \mathbf{B})$. Or $k > 0$, and since $s \stackrel{\leftrightarrow}{\text{dbs}} t$, there are $t_1, \dots, t_{l-1} \in B'$ ($l \in \mathbb{N}$) and $t_l \in \mathbf{B}$ such that $t \rightarrow t_1 \rightarrow \dots \rightarrow t_l$ 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 $t \in \text{split}(B', \mathbf{B})$, again contradicting that $t \in \text{cosplit}(B', \mathbf{B})$.
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 essentially was presented in [9].

$\pi := \pi_0$, i.e., the initial partition;
while π is unstable under some $B \in \pi$
 $\pi :=$ 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

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

The following lemma presents the crucial observation concerning bottom states.

Lemma 3.2. 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 \mathbf{B} of some of the blocks in π iff there is a block $B' \in \pi$ such that

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

Here \subset is meant to be a strict subset.

Proof.

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

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

□

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 $\text{bottom}(B') \cap \text{split}(B', \mathbf{B}) \subset \text{bottom}(B')$ 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 $\text{split}(B', \mathbf{B})$, and those states need to be moved to a new block. This takes time proportional to the number of transitions in B' , i.e., $O(m)$.

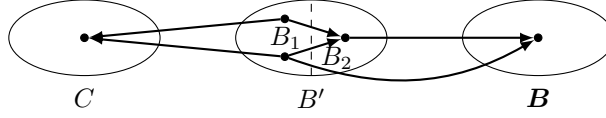


Figure 1: After splitting B' under C , B_1 is not stable under B .

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., $|B| \leq \frac{1}{2} \sum_{B' \in \mathcal{B}} |B'|$, for a block B in a constellation \mathcal{B} . If a block $B' \in \pi$ is unstable under B , then we use a remarkable technique consisting of two procedures running alternately 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 $\mathcal{B} \setminus B^1$ in the splitting without explicitly analysing the states contained therein.

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 the union of one or more blocks from the current partition. If it corresponds with one block, the constellation is called trivial. The current partition is stable with respect to each constellation in \mathcal{C} .

If a constellation \mathcal{B} contains more than one block, we select one block $B \in \mathcal{B}$ which is at most half the size of \mathcal{B} . We check whether the current partition is stable under B and $\mathcal{B} \setminus B$ according to lemma 3.2 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 3.2, we calculate $\text{split}(B', B)$ and $\text{cosplit}(B', B)$, as indicated below.

As noted in [15], $\text{cosplit}(B', B)$ is stable under $\mathcal{B} \setminus B$. Therefore, only further splitting of $\text{split}(B', B)$ under $\mathcal{B} \setminus B$ 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 $\mathcal{B} \setminus B$, which we do not address here explicitly, as it proceeds along the same line.

There is a special data structure to recall for any B' and \mathcal{B} which transitions go from B' to \mathcal{B} . When investigating whether $\text{split}(B', B)$ is stable under B we adapt this list to determine the transitions from $\text{split}(B', B)$ to $\mathcal{B} \setminus B$ and we simultaneously tag the states in B' that have a transition to $\mathcal{B} \setminus B$. Therefore, we know whether there are transitions from $\text{split}(B', B)$ to $\mathcal{B} \setminus B$ and we can traverse the bottom states of $\text{split}(B', B)$ to inspect whether there is a bottom state without a transition to B . Following lemma 3.2, this allows us to determine whether $\text{split}(B', B)$ must be split under $\mathcal{B} \setminus B$ in a time proportional to the size of B . How splitting is carried out is indicated below.

When the current partition has become stable under B and $\mathcal{B} \setminus B$, B is moved from constellation \mathcal{B} into a new trivial constellation B' , and the constellation \mathcal{B} is reduced to contain the states in $\mathcal{B} \setminus B$. Note that the new \mathcal{B} can have become trivial.

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 \mathcal{B} . But if B' is split under block C into B_1 and B_2 , block B_1 is not stable under \mathcal{B} . The reason is, as exemplified by the following lemma,

¹For convenience, we write $\mathcal{B} \setminus B$ instead of $\mathcal{B} \setminus \{B\}$.

that some states that were non-bottom states in B' became bottom states in B_1 .

Lemma 4.1. 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 \mathbf{B} , and $B' \in \pi$ is refined into $B'_1, \dots, B'_k \in \pi'$, 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 \mathbf{B} .

Proof. Assume B'_i is not stable under \mathbf{B} . This means that B'_i is not a subset of \mathbf{B} . Hence, there is a state $s \in B'_i$ such that $s \rightarrow s'$ with $s' \in \mathbf{B}$ and there is a bottom state $t \in B'_i$ with no outgoing transition to a state in \mathbf{B} . But as B' was stable under \mathbf{B} , and s has an outgoing transition to a state in \mathbf{B} , all bottom states in B' must have at least one transition to a state in \mathbf{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 difference, we know that B' must be split further.

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 $B \in \mathbf{B}$. 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 $\mathbf{B} \setminus B$ only has to be applied to $split(B', B)$, or to B' if all bottom states of B' were marked. As noted before $cosplit(B', B)$ is stable under $\mathbf{B} \setminus B$. Define $C := split(B', B)$ or $C := B'$ depending on the situation. We can traverse all bottom states of C and check whether they have outgoing transitions to $\mathbf{B} \setminus B$. This provides us with the blue states. The red states are obtained as we explicitly maintained the list of all transitions from C to $\mathbf{B} \setminus B$. 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 below. After that, it is presented in full detail in section 5. Since it is not trivial how to achieve the $O(m \log n)$ complexity, we have decided to describe the algorithm as detailed as possible.

```

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

```

5 Detailed algorithm

This section presents the data structures, the algorithm to detect which blocks must be split, and the algorithm to split blocks. It follows the outline presented in the previous section.

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 some extra information to the elements in the list. Moreover, for each list L , we maintain pointers to its first and last element, and the size $|L|$.

1. The current partition π consists of a list of blocks. Initially, it corresponds with π_0 . All blocks are part of a single initial constellation C_0 .
2. For each block B , we maintain the following:
 - (a) A reference $B.constln$ to the constellation containing B .
 - (b) A list of the bottom states in B called $B.btm-sts$.
 - (c) A list of the remaining states in B called $B.non-btm-sts$.
 - (d) A list $B.to-constlns$ of structures associated with constellations reachable via a transition from some $s \in B$. Initially, it contains one element associated with C_0 . Each element associated with some constellation 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 $C \setminus B$ (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 $B'.to-constlns$ (which in turn refers back to the element in $B.to-constlns$).
 - 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.
 - (e) A reference $B.inconstln-ref$ is used to refer to the element in $B.to-constlns$ associated with constellation $B.constln$.

Furthermore, when splitting a block B' in constellation B' under a constellation B and block $B \in B$, the following temporary structures are used, with C the new constellation to which B is moved:

- (a) A list $B'.mrkd-btm-sts$ (initially empty) contains states in B' with a transition to B .
 - (b) A list $B'.mrkd-non-btm-sts$ (initially empty) contains states that are marked, but are not bottom states, i.e., each of those states has at least one transition to B and at least one transition to B' .
 - (c) A reference $B'.constln-ref$ is used to refer to the (new) element in $B'.to-constlns$ associated with constellation 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 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. The constellations are represented by two lists. The first one, *non-trivial-constlns*, contains the constellations encompassing two or more blocks of the current partition. The second one, *trivial-constlns*, contains constellations that match a block of the current partition. Initially, if the initial partition π_0 consists of one block, the constellation $C_0 = \{\pi_0\}$ is added to *trivial-constlns* and nothing needs to be done, because the initial partition is already stable. Otherwise C_0 is added to *non-trivial-constlns*.

4. For each constellation B , the following is maintained:
 - (a) A list $B.blocks$ of blocks contained in B . Initially, the only constellation is C_0 .
 - (b) Counter $B.size$ is used to keep track of the number of states in B ; it is equal to $\sum_{B \in \mathcal{B}.blocks} |B|$.
5. Each transition $s \rightarrow s'$ contains its source and target state. Moreover, it refers with *to-constln-cnt* to a variable containing 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 B in the list $B.to-constlns$ where $s \in B$ and $s' \in B$. 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 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 $s.block$ to the block containing s .
 - (b) A static, i.e., not changing during the course of the algorithm, 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 . In the sequel 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 $s.inert-cnt = 0$.
 - (e) Furthermore, when splitting a block B' under B and $B \in \mathcal{B}$, 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 $B \setminus B$.

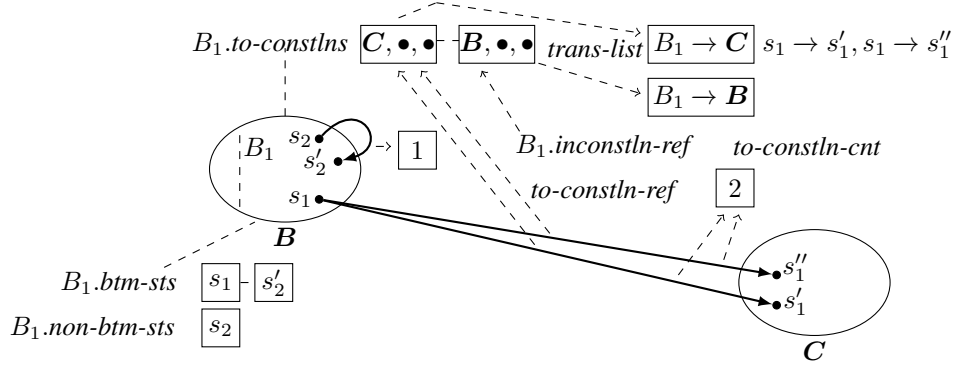


Figure 2: An example showing some of the data structures used in the detailed algorithm.

Figure 2 illustrates some of the used structures. A block B_1 in constellation B contains bottom states s_1, s'_2 and non-bottom state s_2 . State s_1 has two outgoing transitions $s_1 \rightarrow s'_1, s_1 \rightarrow s''_1$ to states s'_1, s''_1 in constellation C . This means that for both transitions, we have the following references:

- (a) *to-constln-cnt* to the number of outgoing transitions from s_1 to C .
- (b) *to-constln-ref* to the element (C, \bullet, \bullet) in $B_1.to-constlns$, where C is the constellation reached by the transitions, and the \bullet 's are the (now uninitialized) references that are used when splitting.
- (c) Via element (C, \bullet, \bullet) , a reference *trans-list* to the list of transitions from B_1 to 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 \mathbf{B} , and that $B_1.inconstln-ref$ refers to the element $(\mathbf{B}, \bullet, \bullet)$ associated with the constellation containing B_1 .

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 \mathbf{B} in *non-trivial-constlns*, and we select a block B from $\mathbf{B}.blocks$ such that $|B| \leq \frac{1}{2}\mathbf{B}.size$. We first check which blocks are unstable for B and $\mathbf{B} \setminus B$.

1. We update the list of constellations w.r.t. B and $\mathbf{B} \setminus B$.
 - (a) Move B into a new constellation \mathbf{C} (with $\mathbf{C}.blocks$ empty and $\mathbf{C}.size = 0$), by removing B from $\mathbf{B}.blocks$, adding it to $\mathbf{C}.blocks$.
 - (b) Add \mathbf{C} to *trivial-constlns*. If $|\mathbf{B}.blocks|=1$, move \mathbf{B} to *trivial-constlns*.
2. Walk through the elements in $B.btm-sts$ and $B.non-btm-sts$, i.e., the states in B . For each state $s \in B$ visit all transitions $s \leftarrow s' \in s.T_{src}$, and do the steps (a) to (d) below for each transition where $B \neq B'$, with B' the block in which s' resides.
 - (a) If $B'.mrkd-btm-sts$ and $B'.mrkd-non-btm-sts$ are empty:
 - i. Put B' in a list *splittable-blks*.
 - ii. Let $B'.coconstln-ref$ refer to $(s \leftarrow s').to-constln-ref$. Let $B'.constln-ref$ refer to a new element in $B'.to-constlns$.
 - (b) If $s'.coconstln-cnt$ is uninitialized, let $s'.constln-cnt$ refer to a new counter (with initial value 0), and let $s'.coconstln-cnt$ refer to $(s \leftarrow s').to-constln-cnt$.
 - (c) If $s' \notin B'.mrkd-btm-sts$ and $s' \notin B'.mrkd-non-btm-sts$, then:
 - i. If s' is a bottom state, move s' from $B'.btm-sts$ to $B'.mrkd-btm-sts$.
 - ii. Else move s' from $B'.non-btm-sts$ to $B'.mrkd-non-btm-sts$.
 - (d) Increment the variable to which $s'.constln-cnt$ refers, and let $(s \leftarrow s').to-constln-cnt$ refer to this variable. Decrement the variable referred to by $s'.coconstln-cnt$. Move $s \leftarrow s'$ from $B'.coconstln-ref.trans-list$ to $B'.constln-ref.trans-list$, and let $(s \leftarrow s').to-constln-ref$ refer to $B'.constln-ref$.
3. Next, check whether B itself can be split. First, mark all states by moving the states in $B.btm-sts$ to $B.mrkd-btm-sts$ and those in $B.non-btm-sts$ to $B.mrkd-non-btm-sts$. Add B to *splittable-blks* and reset references $B.constln-ref$ and $B.coconstln-ref$. Next, for each $s \in B$ visit all transitions $s \rightarrow s' \in s.T_{tgt}$, and do the steps (a) to (c) below for each transition where either $\mathbf{B}' = \mathbf{B}$ or $\mathbf{B}' = \mathbf{C}$, with \mathbf{B}' the constellation in which s' resides.
 - (a) If $\mathbf{B}' = \mathbf{B}$ and $B.constln-ref$ is uninitialized, let $B.coconstln-ref$ refer to $(s \rightarrow s').to-constln-ref$, add a new element for \mathbf{C} to $B.to-constlns$, and let $B.constln-ref$ and $B.inconstln-ref$ refer to this element.
 - (b) If $s.coconstln-cnt$ is uninitialized, let $s.constln-cnt$ refer to a new counter, and $s.coconstln-cnt$ to $(s \rightarrow s').to-constln-cnt$.
 - (c) If $B' = B$, increment the variable to which $s.constln-cnt$ refers, let $(s \rightarrow s').to-constln-cnt$ refer to this variable, and decrement the variable referred to by $s.coconstln-cnt$.
4. Do the steps below for each $B' \in splittable-blks$.
 - (a) If $|B'.btm-sts| > 0$ (there is at least one unmarked bottom state in B'), the block must be split. We leave B' in the list of *splittable-blks*.

- (b) Else, if $B'.coconstln-ref.trans-list$ is not empty, and there is a state $s \in B'.mrkd-btm-sts$ with $s.coconstln-cnt$ uninitialized or 0, the block must be split.
 - (c) Else, no splitting is required. Remove B' from *splittable-blks* and **remove the temporary markings** of B' by doing steps i to iii below.
 - i. Move each $s \in B'.mrkd-btm-sts$ to $B'.btm-sts$ and reset $s.constln-cnt$ and $s.coconstln-cnt$.
 - ii. Move each $s \in B'.mrkd-non-btm-sts$ to $B'.non-btm-sts$. Reset $s.constln-cnt$.
If $s.coconstln-cnt = 0$, delete the variable to which $s.coconstln-cnt$ refers.
Reset $s.coconstln-cnt$.
 - iii. Do the following steps for $ref = constln-ref$ and $ref = coconstln-ref$, if $B'.ref$ is initialized.
 - A. If $|B'.ref.trans-list| = 0$, then first reset $B'.inconstln-ref$ if the element to which $B'.ref$ refers is associated with $B'.constln$, and second remove the element to which $B'.ref$ refers from $B'.to-constlns$ and delete it.
 - B. Reset $B'.ref$.
5. If *splittable-blks* is not empty, start splitting (section 5.3). Else, carry on with finding blocks to split, by selecting another non trivial constellation B and block $B \in \mathcal{B}$, and continuing with step 1. If there are no non trivial constellations left, the current partition is stable, the algorithm terminates.

5.3 Splitting the blocks

Splitting the splittable blocks is performed using the following steps. We walk through the blocks B' in *splittable-blks*, which must be split into two or three blocks under constellation B and block B . For each splitting procedure, we use time proportional to the smallest of the two blocks into which B' is split, where one of the two smallest blocks can have size 0, and we also allow ourselves to traverse the marked states, which is proportional to the time we used to mark the states in the previous step. Create new lists $X_{B'}$, $X_{B''}$, $X_{B'''}$ to keep track of new bottom states when splitting.

If $|B'.btm-sts| = 0$ (all bottom states are marked), then we have $split(B', B) = B'$, and can start with step 3 below.

1. We start to split block B' w.r.t. B . We must determine whether $split(B', B)$ or $cosplit(B', B)$ is the smallest. This is done by performing the following two procedures in lockstep, alternately processing a transition. The entire operation terminates when one of the procedures terminates. If one procedure acquires more states than $\frac{1}{2}|B'|$, it is stopped, and the other is allowed to terminate.
 - (a) The first procedure attempts to collect the states in $split(B', B)$.
 - i. Initialise an empty stack Q and an empty list L . Let D_1 refer to the list consisting of $B'.mrkd-btm-sts$ and $B'.mrkd-non-btm-sts$. In the next step, we walk through the states in D_1 .
 - ii. Perform **detect1** as long as $|L| \leq \frac{1}{2}|B'|$:
 - A. While Q is not empty or we have not walked through all states in D_1 , do the following steps.
 - If Q is empty, push the next state in D_1 on Q and add it to L .
 - Pop s from Q . For all $s \leftarrow s' \in s.T_{src}$ if $s' \in B'$ and $s' \notin L$, add s' to L and push s' on Q .
 - (b) The second procedure attempts to collect the states in $cosplit(B', B)$. It uses a priority queue P , in which the priority of a state s represents the number of outgoing inert transitions to a target state for which it has not yet been determined whether it is in $cosplit(B', B)$. If the priority of a state s becomes 0, it is obvious that s must be in $cosplit(B', B)$.
 - i. Create an empty priority queue P and an empty list L' . Let D_2 refer to the list $B'.btm-sts$.
 - ii. Perform **detect2** as long as $|L'| \leq \frac{1}{2}|B'|$:
 - A. While P has states with priority 0 or we have not walked through all states in D_2 , do the following steps.

- If we have not walked through all states in D_2 , let s be the next state in D_2 . Else, get a state with priority 0 from P , and let s be that state. Add s to L' .
 - For all $s \leftarrow s' \in s.T_{src}$, do the following steps.
 - If $s' \in B'$, $s' \notin P \cup L'$, and $s' \notin B'.mrkd-non-btm-sts$ (or s' does not have a transition to $B \setminus B'$; this last condition is required when **detect2** is invoked in 5.3.4.b and 5.3.7.b.i.B, and can be checked for $s' \in mrkd-non-btm-sts$ by determining whether the variable to which $s'.coconstln-cnt$ refers, minus $s'.inert-cnt$ if $B' = B$, is larger than 0. Else, it can be checked by walking over the transitions $s' \rightarrow s'' \in s.T_{tgt}$), add s' with priority $s'.inert-cnt$ to P .
 - If $s' \in P$, decrement the priority of s' in P .
2. The next step is to actually carry out the splitting of B' . Create a new block B'' with empty lists, and add it to the list of blocks. Set $B''.constln$ to $B'.constln$, and add B'' to the list of blocks of that constellation.

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 B'' . Below we refer to this list as N . For each $s \in N$, do the following:

- (a) Set $s.block$ to B'' , and move s from the list in which it resides in B' to the corresponding list in B'' .
 - (b) For each $s \rightarrow s' \in T_{tgt}$, do the following steps.
 - i. If $(s \rightarrow s').to-constln-ref$ is initialized, i.e., $s \rightarrow s'$ is not inert, consider the list element l in $B'.to-constlns$ retrievable by $(s \rightarrow s').to-constln-ref$. Check whether there is a corresponding new element in $B''.to-constlns$. If so, l refers to this new element, which we call l' . If not, create it, add it to $B''.to-constlns$ and call it also l' , set the constellation in this new l' to that of l , set $B''.inconstln-ref$ to l' in case this constellation is $B''.constln$, set $B''.constln-ref$ to l' in case l refers to $B'.constln-ref$, set $B''.coconstln-ref$ to l' in case l refers to $B'.coconstln-ref$, and let l and l' refer to each other. Move $s \rightarrow s'$ from $l.trans-list$ to $l'.trans-list$ and let $(s \rightarrow s').to-constln-ref$ refer to l' .
 - ii. Else, if $s' \in B' \setminus N$ (an inert transition becomes non-inert):
 - A. Decrement $s.inert-cnt$.
 - B. If $s.inert-cnt=0$, add s to $X_{B''}$, move s from $B''.non-btm-sts$ or $B''.mrkd-non-btm-sts$ to the corresponding bottom states list in B'' . If $B''.inconstln-ref$ is uninitialized, create a new element for $B''.constln$, add it to $B''.to-constlns$, let $B''.inconstln-ref$ refer to that element, and if $B'.inconstln-ref$ is initialized, let $B'.inconstln-ref$ and $B''.inconstln-ref$ refer to each other. Add $s \rightarrow s'$ to $B''.inconstln-ref.trans-list$ and let $(s \rightarrow s').to-constln-ref$ refer to $B''.inconstln-ref$.
 - (c) For each $s \leftarrow s' \in T_{src}$, $s' \in B' \setminus N$ (an inert transition becomes non-inert):
 - i. Decrement $s'.inert-cnt$.
 - ii. If $s'.inert-cnt=0$, add s' to $X_{B'}$, and move s' from $B'.non-btm-sts$ or $B'.mrkd-non-btm-sts$ to the corresponding bottom states list in B' . If $B'.inconstln-ref$ is uninitialized, create a new element for constellation $B'.constln$ and add it to $B'.to-constlns$, let $B'.inconstln-ref$ refer to that element, and if $B''.inconstln-ref$ is initialized, let $B'.inconstln-ref$ and $B''.inconstln-ref$ refer to each other. Add $s \leftarrow s'$ to $B'.inconstln-ref.trans-list$ and let $(s \leftarrow s').to-constln-ref$ refer to $B'.inconstln-ref$.
3. For each element l in $B''.to-constlns$ referring to an element l' in $B'.to-constlns$, do the following steps.
- (a) If $l'.trans-list$ is empty and l' does not refer to $B'.constln-ref$ and not to $B'.coconstln-ref$, reset $B'.inconstln-ref$ if l' is associated with $B'.constln$, remove l' from $B'.to-constlns$ and delete it.
 - (b) Else, reset the reference from l' to l .

- (c) Reset the reference from l to l' .
4. Next, we must consider splitting $split(B', B)$ under $B \setminus B$, or if B' was stable under B , we must split B' under $B \setminus B$. Define $C = split(B', B)$. If B' was not split, then $C = B'$. C is stable under $B \setminus B$ if $C.coconstln-ref$ is uninitialized or $C.coconstln-ref.trans-list$ is empty or for all $s \in C.mrkd-btm-sts$ it holds that $s.coconstln-cnt > 0$. If this is not the case, then we must determine which of the blocks $split(C, B \setminus B)$ or $cosplit(C, B \setminus B)$ is the smallest in a time proportional to the smallest of the two. This is again done by simultaneously iterating over the transitions of states in both sets in lockstep. The entire operation terminates when one of the two procedures terminates. If one of the procedures acquires more than $\frac{1}{2}|C|$ states, that procedure is stopped, and the other is allowed to terminate.
- (a) The first procedure attempts to collect the states in $split(C, B \setminus B)$.
- i. Create an empty stack Q and an empty list L . Let D_1 be the list of states s occurring in some $s \rightarrow s'$ in the list $split(B', B).coconstln-ref.trans-list$ (in practice we walk over D_1 by walking over the latter list).
 - ii. Perform **detect1** with $B' = C$.
- (b) The second procedure attempts to collect the states in $cosplit(C, B \setminus B)$.
- i. Create an empty priority queue P and an empty list L' . Let D_2 be the list of states s with $s.coconstln-cnt = 0$ in $C.mrkd-btm-sts$ (in practice we walk over the latter list and check the condition).
 - ii. Perform **detect2** with $B' = C$.

Finally, we split C by moving either $split(C, B \setminus B)$ or $cosplit(C, B \setminus B)$ to a new block B''' , depending on which of the two is the smallest. This can be done by using the procedure described in steps 2 and 3, where in step 2.b.ii.B, we fill a state list $X_{B'''}$ instead of $X_{B''}$, and in 2.c.ii, we possibly add states to X_C instead of $X_{B'}$ (we define $X_C = X_{B'}$ if $C = B'$, otherwise $X_C = X_{B''}$). We move all states in X_C that are now in B''' to $X_{B'''}$.

5. Remove the temporary markings of each block C resulting from the splitting of B' (see steps 5.2.4.c.i to iii).
6. If the splitting of B' resulted in new bottom states (either $X_{B'}$, $X_{B''}$, or $X_{B'''}$ is not empty), check for those states whether further splitting is required. This is the case if from some new bottom states, not all constellations can be reached which can be reached from the block. Perform the following for $\hat{B} = B', B'',$ and B''' , and all $s \in X_{\hat{B}}$:
- (a) For all $s \rightarrow s' \in T_{tgt}$ ($s' \in B'$):
- i. If it does not exist, create an empty list $S_{B'}$ and associate it with B' in $\hat{B}.to-constlns$ (accessible via $(s \rightarrow s').to-constln-ref$), and move B' to the front of $\hat{B}.to-constlns$.
 - ii. If $s \notin S_{B'}$, add it.
- (b) Move s from $X_{\hat{B}}$ to $\hat{B}.new-btm-sts$.
7. Check if there are unstable blocks that require further splitting, and if so, split further. Repeat this until no further splitting is required. A stack Q' is used to keep track of the blocks that require checking. For $\hat{B} = B', B'',$ and B''' , push \hat{B} on Q' if $|\hat{B}.new-btm-sts| > 0$. While Q' is not empty, perform steps a to c below.
- (a) Pop block \hat{B} from Q' .
- (b) Find a constellation under which \hat{B} is not stable by walking through the $B \in \hat{B}.to-constlns$. If $|S_B| < |\hat{B}.new-btm-sts|$, then further splitting is required under B :
- i. Find the smallest subblock of \hat{B} by performing the following two procedures in lockstep.
 - A. The first procedure attempts to collect the states in $split(\hat{B}, B)$.

- Create an empty stack Q and an empty list L . Let D_1 be the list of states s occurring in some $s \rightarrow s'$ with $s' \in \mathbf{B}$ in the list *trans-list* associated with $\mathbf{B} \in \hat{B}.to\text{-constlns}$ (in practice we walk over D_1 by walking over *trans-list*).
 - Perform **detect1** with $B' = \hat{B}$.
- B. The second procedure attempts to collect the states in $\text{cosplit}(\hat{B}, \mathbf{B})$.
- Create an empty priority queue P and an empty list L' . Let D_2 be the list of states $s \in \hat{B}.new\text{-btm-sts} \setminus S_{\mathbf{B}}$.
 - Perform **detect2** with $B' = \hat{B}$.
- ii. Continue the splitting of \hat{B} by performing step 2 to produce a new block \hat{B}' and a list of new bottom states $X_{\hat{B}'}$. Move all states in $\hat{B}.new\text{-btm-sts}$ that have moved to \hat{B}' to $\hat{B}'.new\text{-btm-sts}$. Update the $S_{\mathbf{B}}$ lists by walking over the $l \in \hat{B}.to\text{-constlns}$ and doing the following steps as long as an empty $S_{\mathbf{B}}$ is not encountered. Note that the l still refer to corresponding elements l' in $\hat{B}'.to\text{-constlns}$.
- A. For all $s \in S_{\mathbf{B}}$, if $s \in \hat{B}'$, then insert s in the $S_{\mathbf{B}}$ associated with l' (if this list does not exist yet, create it, and move l' to the front of $\hat{B}'.to\text{-constlns}$) and remove s from the $S_{\mathbf{B}}$ associated with l .
- B. If the $S_{\mathbf{B}}$ of l is now empty, remove it, and move l to the back of $\hat{B}.to\text{-constlns}$.
- iii. Perform step 3 for elements l' in $\hat{B}'.to\text{-constlns}$ referring to an element l in $\hat{B}.to\text{-constlns}$. Skip steps 4 and 5, and continue with step 6 for $\hat{B}, \hat{B}', X_{\hat{B}}$ and $X_{\hat{B}'}$.
- iv. Push \hat{B} on Q' if $|\hat{B}.new\text{-btm-sts}| > 0$ and \hat{B}' on Q' if $|\hat{B}'.new\text{-btm-sts}| > 0$.
- (c) If no further splitting was required for \hat{B} , empty $\hat{B}.new\text{-btm-sts}$ and remove the remaining $S_{\mathbf{B}}$ associated with constellations $\mathbf{B} \in \hat{B}.to\text{-constlns}$.
8. If $B'.constln \in \text{trivial-constlns}$, 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).

Definition 6.1 (Labeled transition system). A labeled transition system (LTS) is a three tuple $A = (S, Act, \rightarrow)$ where

1. S is a finite set of *states*. The number of states is generally denoted by n .
2. Act is a finite set of actions including the *internal action* τ .
3. $\rightarrow \subseteq S \times Act \times S$ is a *transition relation*. The number of transitions is generally denoted as by m .

It is common to write $t \xrightarrow{a} t'$ for $(t, a, t') \in \rightarrow$.

There are various, but equivalent, ways to define branching bisimulation. We use the definition below.

Definition 6.2 (Branching bisimulation). Consider the labeled transition system $A = (S, Act, \rightarrow)$. We call a symmetric relation $R \subseteq S \times S$ a *branching bisimulation relation* iff for all $s, t \in S$ such that $s R t$, the following conditions hold for all actions $a \in Act$:

1. If $s \xrightarrow{a} s'$, then
 - (a) Either $a = \tau$ and $s' R t$, or
 - (b) There is a sequence $t \xrightarrow{\tau} \dots \xrightarrow{\tau} t'$ of (zero or more) τ -transitions such that $s R t'$ and $t' \xrightarrow{a} t''$ with $s' R t''$.

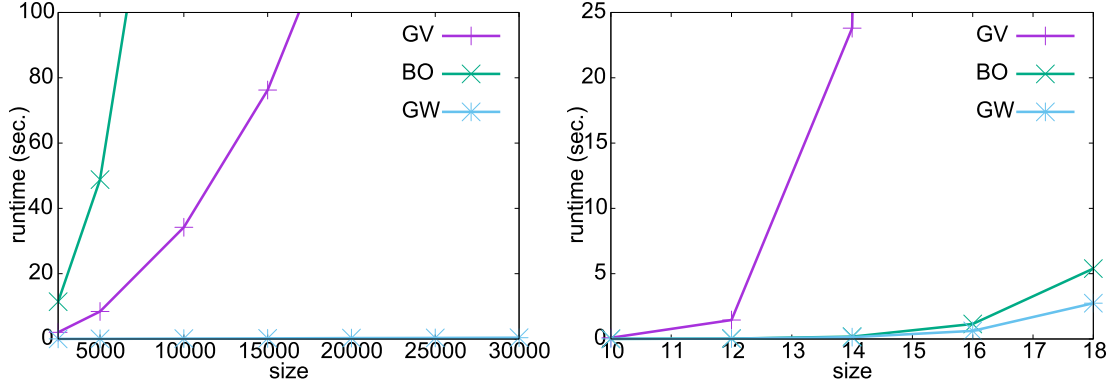


Figure 3: Runtime results for $(a \cdot \tau)^{size}$ sequences (left) and trees of depth $size$ (right)

Two states s and t are *branching bisimilar* iff there is a branching bisimulation relation R such that $s R t$.

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

Definition 6.3 (LTS embedding). 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. $S_A = S \cup \{\langle a, t \rangle \mid s \xrightarrow{a} t \text{ for some } t \in S\}$.
2. $AP = Act \cup \{\perp\}$.
3. \rightarrow is the least relation satisfying $(s, t \in S, a \in Act \setminus \tau)$:

$$\frac{s \xrightarrow{a} t}{s \rightarrow \langle a, t \rangle} \quad \frac{}{\langle a, t \rangle \rightarrow t} \quad \frac{s \xrightarrow{\tau} t}{s \rightarrow t}$$

4. $L(s) = \{\perp\}$ for $s \in S$ and $L(\langle a, t \rangle) = \{a\}$.

The following theorem stems from [14].

Theorem 6.4. 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 requires $O(m \log(n+m))$ time. As m is at most $|Act|n^2$ this is also equal to $O(m(\log |Act| + \log n))$.

A final note is that the algorithm can also easily be adapted to determine divergence-sensitive branching bisimulation [8], by simply adding a self loop indicating divergence to those states on a τ -loop, similar to the way stuttering equivalence is calculated using divergence-blind stuttering equivalence.

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 & 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 section 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 designed 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 figure 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 . This example is particularly suitable for BO which 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 set² and the mCRL2 toolset³. For each case, the best runtime result has been highlighted in bold. Some characteristics of each case are given on the left, in particular the number of states (n) and transitions (m) in the original LTS and the number of states ($min. n$) and transitions ($min. m$) in the minimized LTS.

The final two cases stem from mCRL2 models distributed with the mCRL2 toolset as follows:

- **dining_14** is an extension of the Dining Philosophers model to fourteen philosophers;
- **1394-fin3** extends the 1394-fin model to three processes and two data elements.

The 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.

²<http://cadp.inria.fr/resources/vlts>.

³<http://www.mcrl2.org>.

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

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

References

- [1] A. Aho, J. Hopcroft and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [2] S.C. Blom and S. Orzan. Distributed Branching Bisimulation Reduction of State Spaces. In FMICS'03, ENTCS 80, pp. 109-123. Elsevier, 2003.
- [3] S.C. Blom and J.C. van de Pol. Distributed Branching Bisimulation Minimization by Inductive Signatures. In PDMC'09, EPTCS 14, pp. 32-46. Open Publ. Association, 2009.
- [4] M.C. Browne, E.M. Clarke and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science* 59(1,2):115-131, 1988.
- [5] K. Chatterjee and M. Henzinger. Faster and Dynamic Algorithms for Maximal End-Component Decomposition and Related Graph Problems in Probabilistic Verification. In SODA'11, pp. 1318-1336. SIAM, 2011.
- [6] S. Cranen, J.F. Groote, J.J.A. Keiren, F.P.M. Stappers, E.P. de Vink, J.W. Wesselink, and T.A.C. Willemse. An overview of the mCRL2 toolset and its recent advances. TACAS'13, LNCS 7795, pp. 199-213, Springer, 2013. See also www.mcrl2.org.
- [7] H. Garavel, F. Lang, R. Mateescu and W. Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *Software Tools for Technology Transfer* 15(2):98-107, 2013.
- [8] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM* 43(3):555-600, 1996.
- [9] J.F. Groote and F.W. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In ICALP'90, LNCS 443, pp. 626-638. Springer, 1990.
- [10] J.F. Groote and M.R. Mousavi. *Modeling and Analysis of Communicating Systems*. The MIT Press, 2014.
- [11] P. Kannelakis, S. Smolka. CCS Expressions, Finite State Processes and Three Problems of Equivalence. *Information and Computation* 86:43-68, 1990.
- [12] W. Li. Algorithms for Computing Weak Bisimulation Equivalence. In TASE'09, pp. 241-248. IEEE, 2009.
- [13] R. Milner. *Calculus of Communicating Systems*. Lecture Notes in Computer Science 92. Springer-Verlag 1980.
- [14] R. De Nicola and F.W. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM* 42:458-487, 1995.
- [15] R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computation* 16(6):973-989, 1987.
- [16] F. Ranzato and F. Tapparo. Generalizing the Paige-Tarjan Algorithm by Abstract Interpretation. *Information and Computation* 206(5):620-651, 2008.
- [17] M.A. Reniers, R. Schoren, and T.A.C. Willemse. Results on embeddings between state-based and event-based systems. *The Computer Journal* 57(1):73-92, 2014.
- [18] H. Virtanen, H. Hansen, A. Valmari, J. Nieminen and T. Erkkilä. Tampere Verification Tool. In TACAS'04, LNCS 2988, pp. 153-157. Springer, 2004.
- [19] A.J. Wijs. GPU Accelerated Strong and Branching Bisimilarity Checking. In TACAS'15, LNCS 9035, pp. 368-383. Springer, 2015.