A simpler O (m log n) algorithm for branching bisimilarity on labelled transition systems

Technische Universiteit Eindhoven
Department of Mathematics and Computer Science

A simpler $O(m \log n)$ algorithm for branching bisimilarity on labelled transition systems

David N. Jansen, Jan Friso Groote, Jeroen J.A. Keiren, Anton Wijs

19/03

ISSN 0926-4515

editor: prof.dr.ir. J.J. van Wijk

Reports are available at:
https://research.tue.nl/en/publications/?search=Computer+science+reports+eindhoven&originalSearch=Computer+science+reports+eindhoven&pageSize=50&ordering=publicationYearThenTitle&descending=true&showAdvanced=false&allConcepts=true&inferConcepts=true&searchBy=RelatedConcepts

Computer Science Reports 19-03
Eindhoven, September 2019

A simpler $O(m \log n)$ algorithm for branching bisimilarity on labelled transition systems

David N. Jansen*
Institute of Software, Chinese Academy of Sciences, Beijing, China
and Sino-Europe Institute of Dependable Smart Software

Jan Friso Groote* Jeroen J.A. Keiren Anton Wijs
Department of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands

dnjansen@ios.ac.cn, {J.F.Groote, J.J.A.Keiren, A.J.Wijs}@tue.nl

Abstract

Branching bisimilarity is a behavioural equivalence relation on labelled transition systems that takes internal actions into account. It has the traditional advantage that algorithms for branching bisimilarity are more efficient than all algorithms for other weak behavioural equivalences, especially weak bisimilarity. With m the number of transitions and n the number of states, the classic $O(mn)$ algorithm has recently been replaced by an $O(m \log n)$ algorithm, which is unfortunately rather complex. This paper combines the ideas from Groote et al. [9] with the ideas from Valmari [19]. This results in a simpler $O(m \log n)$ algorithm. Benchmarks show that this new algorithm is faster and more memory efficient than all its predecessors.

1 Introduction

Branching bisimilarity [8] is an alternative to weak bisimilarity [16]. Both equivalences allow the reduction of labelled transition systems containing transitions labelled with internal actions, which are also referred to as silent, hidden or τ-actions.

One of the distinct advantages of branching bisimilarity is that, from the outset, an efficient algorithm has been available [10]. This algorithm can be used to calculate whether two states in a labelled transition system are equivalent, and to calculate a quotient transition system. The algorithm had complexity $O(mn)$ with m the number of transitions and n the number of states. It is more efficient than classic algorithms for weak bisimilarity, which use transitive closure (for instance, [15] runs in $O\big(n^2 m \log n + mn^{2.376}\big)$, where $n^{2.376}$ is the time for computing the transitive closure), and algorithms for weak simulation (the strong simulation relation can be computed in $O(mn)$ [12], and for weak simulation first the transitive closure of the transition relation needs to be computed). The algorithm is also far more efficient than algorithms for trace-based equivalence notions, such as (weak) trace equivalence or weak failure equivalence, as these are generally PSPACE-complete on finite-state labelled transition systems [15].

Branching bisimilarity is interesting in several other respects. Not only is it a useful notion to compare the behaviour of labelled transition systems directly, as it exactly respects the branching structure of behaviour, it also enjoys a number of nice mathematical properties such as the existence of a canonical quotient with a minimal number of states and transitions modulo branching bisimilarity (contrary to, for instance, trace-based equivalences). Additionally, as branching bisimilarity is coarser than virtually any other conceivable behavioural equivalence taking internal actions into account [7], it is ideal for preprocessing. In order to calculate a desired equivalence,

*This work is partly done during a visit of the first author at Eindhoven University of Technology, and a visit of the second author at the Institute of Software, Chinese Academy of Sciences.

one can first reduce the behaviour modulo branching bisimilarity, before applying a dedicated algorithm on the often substantially reduced transition system. In the mCRL2 toolset [5] this is common practice.

In [9, 11] an algorithm to calculate stuttering equivalence on Kripke structures with complexity $O(m \log n)$ was proposed. Stuttering equivalence essentially differs from branching bisimilarity in the fact that transitions do not have labels and as such all transitions can be viewed as internal. In these papers it was shown that branching bisimilarity can be calculated by translating labelled transition systems to Kripke structures, encoding the labels of transitions into labelled states following [6, 18]. This led to an $O(m \log n)$ algorithm for branching bisimilarity.[1]

Unfortunately, the algorithm in [9, 11] has two disadvantages. First, the translation to Kripke structures introduces a new state and a new transition per action label and target state of a transition, and as such increases the memory required to calculate branching bisimilarity substantially, depending on the structure of the transition system. This made it far less memory efficient than the classical algorithm of [10], and this was actually perceived as a substantial practical hindrance. For instance, when reducing systems consisting of tens of millions of states, such as [2], memory consumption is the bottleneck of the algorithm from [9, 11]. Second, the algorithm in [9, 11] is very complex. To realise the targeted $O(m \log n)$ complexity, several subtle situations that can occur while running the algorithm were handled using dedicated subalgorithmic steps. To illustrate the complexity, implementing the algorithm of [9, 11] took approximately half a man-year.

Contributions. We present an algorithm for branching bisimilarity that runs directly on labelled transition systems in $O(m \log n)$ time and that is simpler than the algorithm of [9, 11].

To achieve this we use an idea from Valmari and Lehtinen [19, 20], which they apply in the context of strong bisimulation. They observed that the standard Paige-Tarjan algorithm [17], which has $O(m \log n)$ time complexity for strong bisimilarity on Kripke structures, has a slightly higher time complexity, namely $O(m \log m)$, when applied to labelled transition systems. Valmari [19] solves this in a sophisticated way by introducing a partition of transitions, whose elements he (and we) calls *bunches.*

Using this idea we design our more straightforward algorithm for branching bisimilarity on labelled transition systems. Essentially, this makes the maintenance of action labels particularly straightforward and allows to simplify stability reassessment in case of new bottom states. It also leads to a novel main invariant, which we formulate as Invariant 3.2. It allows us to prove the correctness of the algorithm in a far more straightforward way than before.

To simplify the complexity notations we assume that $n \leq m + 1$. This is not a significant restriction, since it is satisfied by any labelled transition system in which every non-initial state has an incoming transition. This can easily be achieved by preprocessing the graph to remove unreachable states. Furthermore, we assume that we can access action labels fast enough to bucket sort the transitions in time $O(m)$, which is for instance the case if the action labels are consecutively numbered.

We provide a detailed proof of correctness of the algorithm and show using benchmarks that this new algorithm outperforms all preceding algorithms both in time and space when the labelled transition systems become sizeable. This is illustrated with more than 30 example LTSs.

Despite the fact that this new algorithm is more straightforward than the previous $O(m \log n)$ algorithm [9], the implementation of the algorithm into code is still not easy. To guard against implementation errors, we extensively applied random testing, comparing the output with that of other algorithms. The algorithms and their source code are freely available for use in the mCRL2 toolset [5].

Historical overview. For those new to the area of algorithms for bisimulation equivalences on labelled transition systems, it might be useful to review the major concepts that have been developed in this field throughout the years. Following Kanellakis and Smolka [14], efficient

[1] The analysis in [9] claims a complexity of $O(m(\log |Act| + \log n))$. A more careful analysis, however, shows the actual complexity to be $O(m \log n)$, see Appendix A.

algorithms use partition refinement. The states of the transition system are partitioned into blocks, such that equivalent states are in the same block. These blocks are refined until non-equivalent states are in different blocks. The original algorithm [14] calculated strong bisimilarity and had time complexity $O(mn)$. The main idea is to find a *splitter,* i.e., a block that shows that some states are not equivalent, and then move these states to separate blocks.

Subsequently, the seminal article of Paige and Tarjan [17] presented an efficient algorithm for strong bisimulation minimisation of Kripke structures. Its main data structure consists of two partitions, a fine one into blocks and a coarse one into (what is called in [9]) *constellations* of blocks. The fine partition stores the current knowledge about inequivalence of states, and the coarse partition stores the current knowledge on which blocks cannot act as splitters. When the two partitions coincide, no more splitters exist, so the blocks in the fine partition are the bisimulation equivalence classes. Paige and Tarjan's algorithm repeatedly splits a constellation with multiple blocks into two parts and splits the fine partition if the new constellations actually lead to some splits. Their ingenious data structures ensure Hopcroft's "Process the smaller half" principle [13], guaranteeing that the work done requires time proportional to the *smaller* of the splitters. This leads to a time complexity in $O(m \log n)$ on Kripke structures.

The next key insight was by Valmari, who introduced the idea to use a partition of transitions into bunches [19] to store the knowledge about non-splitters. This resulted in an $O(m \log n)$ algorithm for strong bisimilarity on labelled transition systems, even though m may be larger than n^2 when there are enough transition labels.

For branching bisimilarity, Groote and Vaandrager [10] presented an algorithm with worst-case time complexity in $O(mn)$. They established that in order to determine that a block can be split it is only necessary to look at its *bottom states,* i.e., states that have no outgoing internal transition in the same block. In addition to the algorithm of [14] the algorithm of Groote and Vaandrager [10] had to determine which states can reach certain bottom states via internal transitions in the block to split that block. They also observed that stability is not preserved under splitting when new bottom states emerge. Therefore, stability of existing blocks had to be reassessed by the algorithm.

More than 25 years later, [9, 11] managed to merge the ideas of Paige and Tarjan with those of Groote and Vaandrager, finding an algorithm for stuttering equivalence that has time complexity $O(m \log n)$ on Kripke structures as well as time complexity $O(m \log n)$ on LTSs using a translation of LTSs to Kripke structures. The first essential difficulty that had to be overcome was that calculating the reachability of states through internal transitions must be done in time proportional to the smallest block that is split off, following the "Process the smaller half" principle. This was done by two coroutines that are executed in parallel to identify those states that can reach certain bottom states via internal transitions, and simultaneously identify the states from which those bottom states are not reachable. As soon as the smaller set of states was found, the other coroutine was terminated. The other essential contribution of [9, 11] is that reassessing stability of the partition can be done in time proportional to $\log n$ times the number of (incoming and outgoing) transitions of new bottom states. As each state becomes a bottom state at most once, this fits into the time bound. Dealing with this was done in a rather delicate post-processing stage that would be executed whenever a new bottom state was found. We present an algorithm without postprocessing.

Overview of the article. In Section 2 we provide the definition of labelled transition systems and branching bisimilarity. In Section 3 we provide the core algorithm with high-level data structures, correctness and complexity. The next section presents the procedure for splitting blocks, which can be presented as an independent pair of coroutines. Section 5 provides the details of the algorithm, especially how the high-level data structures can be represented efficiently. Section 6 presents some benchmarks.

2 Branching bisimilarity

In this section we define labelled transition systems and branching bisimilarity.

Definition 2.1 (Labelled transition system)**.** A *labelled transition system* (LTS) is a triple $A = (S, Act, \rightarrow)$ where

1. S is a finite set of *states.* The number of states is 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 necessarily finite and denoted by m.

It is common to write $t \xrightarrow{a} t'$ for $(t, a, t') \in \rightarrow$. Using a slight abuse of notation we write $t \xrightarrow{a} t' \in T$ instead of $(t, a, t') \in T$ for $T \subseteq \rightarrow$. We also write $t \xrightarrow{a} T$ for the set of transitions $\{t \xrightarrow{a} t' \mid t' \in T\}$, and likewise $T \xrightarrow{a} T'$ for the set $\{t \xrightarrow{a} t' \mid t \in T \text{ and } t' \in T'\}$. We refer to all actions except τ as the visible actions. The transitions labelled with τ are the invisible or hidden transitions. If $t \xrightarrow{a} t'$, we say that from t, the state t', the action a, and the transition $t \xrightarrow{a} t'$ are *reachable.*

Definition 2.2 (Branching bisimilarity)**.** Let $A = (S, Act, \rightarrow)$ be a labelled transition system. We call a relation $R \subseteq S \times S$ a *branching bisimulation relation* iff it is symmetric and for all $s, t \in S$ such that $s \; R \; t$ and all transitions $s \xrightarrow{a} s'$ we have:

1. $a = \tau$ and $s' \; R \; t$, or
2. there is a sequence $t \xrightarrow{\tau} \cdots \xrightarrow{\tau} t' \xrightarrow{a} t''$ such that $s \; R \; t'$ and $s' \; R \; t''$.

Two states s and t are *branching bisimilar,* denoted by $s \leftrightarrows_b t$, iff there is a branching bisimulation relation R such that $s \; R \; t$.

Note that branching bisimilarity is an equivalence relation. Given an equivalence relation R, a transition $s \xrightarrow{a} t$ is called *inert* iff $a = \tau$ and $s \; R \; t$. If $t \xrightarrow{\tau} t_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} t_{n-1} \xrightarrow{\tau} t_n \xrightarrow{a} t'$ such that $t \; R \; t_i$ for $1 \leq i \leq n$, we say that the state t_n, the action a, and the transition $t_n \xrightarrow{a} t'$ are *inertly reachable* from t.

3 The algorithm

We now present our algorithm to calculate branching bisimilarity at an abstract level and assign time budgets, i.e., indications of how much time each step is allowed to take. The details of the implementation, which are essential to fit the time budgets, are given in Section 5. We first describe the basic data structures and subsequently the algorithm, its correctness and complexity. The algorithm depends on a block splitting procedure, which is explained in Section 4. In this and the following sections, we use the labelled transition system $A = (S, Act, \rightarrow)$.

3.1 The essential data types

The algorithm relies heavily on partitioning sets, especially, sets of states and sets of transitions.

Definition 3.1 (Partition)**.** For a set X a partition Π of X is a disjoint cover of X, i.e., $\Pi = \{B_i \subseteq X \mid B_i \neq \emptyset, 1 \leq i \leq k\}$ such that $B_i \cap B_j = \emptyset$ for all $1 \leq i < j \leq k$ and $X = \bigcup_{1 \leq i \leq k} B_i$.

A partition Π' is a *refinement* of Π iff for every $B' \in \Pi'$ there is some $B \in \Pi$ such that $B' \subseteq B$.

Note that a partition induces an equivalence relation in the following way: $s \equiv_\Pi t$ iff there is some $B \in \Pi$ containing both s and t.

The algorithm uses two main partitions. Partition Π_s is a partition of states in S that is coarser than branching bisimilarity. We refer to the elements of Π_s as *blocks,* for which we typically use the letter B. Partition Π_t partitions the non-inert transitions of $\rightarrow$, where inertness of τ-transitions

is interpreted with respect to $\equiv_{\Pi_s}$. We refer to the elements of Π_t as *bunches,* for which we use the letter T.

In the algorithm, the partition of blocks Π_s records the current knowledge about branching bisimilarity: two states are in different blocks iff the algorithm has found a proof that they are not branching bisimilar (see Invariant 3.6). The partition of transitions Π_t records the current knowledge about splitters. For each bunch of transitions, we initially assume that they can pairwise simulate each other (i.e., they can serve as transitions $s \xrightarrow{a} s'$ and $t' \xrightarrow{a} t''$ in Definition 2.2), until proven otherwise. The algorithm maintains the invariant (formalised in Invariant 3.2) that, whenever a state in a block has a transition in a bunch, then every state in that block can inertly reach a transition in the same bunch, such that the condition of Definition 2.2 is satisfied. Whenever this invariant is satisfied for a particular block and bunch, we say that the block is *stable* with respect to that bunch.

However, Definition 2.2 comes with some constraints on transition $t' \xrightarrow{a} t''$: first, it needs to have the same action label as $s \xrightarrow{b} s'$, and second, s' and t'' need to be in the same target block. If these constraints are violated, it appears that our initial assumption about the bunch was incorrect, and we try to correct this by splitting the bunch into parts that do satisfy the constraints. We do this by splitting off an *action-block-slice,* a subset of transitions in a bunch with the same action label and the same target block and placing it in a new bunch. The new bunch is called the *primary* splitter, the remainder of the bunch from which it was split off is called the *secondary* splitter. After such a split, the blocks in Π_s need to be split with respect to both splitters to re-establish the invariant.

From the two partitions Π_s and Π_t, we infer the following derived sets that are used in the algorithm. For bunches $T \in \Pi_t$, actions $a \in Act$ and blocks $B, B' \in \Pi_s$, we have:

- The *block-bunch-slices,* i.e., the transitions in T that start in B: $T_{B\to} = \{s \xrightarrow{b} s' \in T \mid s \in B\}$.
- The *action-block-slices,* i.e., the transitions in T that have label a and end in B': $T_{\xrightarrow{a}B'} = \{s \xrightarrow{a} s' \in T \mid s' \in B'\}$.
- A block-bunch-slice intersected with an action-block-slice:
  $T_{B\xrightarrow{a}B'} = T_{B\to} \cap T_{\xrightarrow{a}B'} = \{s \xrightarrow{a} s' \in T \mid s \in B \wedge s' \in B'\}$.
- The *bottom states* of a block, i.e., the states without an outgoing inert transition:
  $Bottom(B) = \{s \in B \mid \neg\exists s' \in B. s \xrightarrow{\tau} s'\}$.
- The states in a block with a transition in a bunch: $B_{\xrightarrow{T}} = \{s \mid s \xrightarrow{a} s' \in T_{B\to}\}$.
- The blocks splittable by an action-block-slice:
  $splittableBlocks(T_{\xrightarrow{a}B'}) = \{B \in \Pi_s \mid \emptyset \subset T_{B\xrightarrow{a}B'} \subset T_{B\to}\}$.
- The number of action-block-slices contained in a bunch:
  $\#aB'(T) = |\{T_{\xrightarrow{a}B'} \mid a \in Act, B' \in \Pi_s\} \setminus \{\emptyset\}|$.
- If $B, B' \in \Pi_s$ and $B \neq B'$, then $B \xrightarrow{\tau} B'$ are the non-inert τ-transitions between B and B'.
- The outgoing transitions of a block: $B_{\to} = \{s \xrightarrow{a} s' \mid s \in B, a \in Act \text{ and } s' \in S\}$.
- The incoming transitions of a block: $B_{\leftarrow} = \{s \xrightarrow{a} s' \mid s \in S, a \in Act \text{ and } s' \in B\}$.

The first two of these sets (block-bunch-slices and action-block-slices) are maintained as auxiliary data structures in the algorithm in order to meet the required performance bounds. If the partitions Π_s or Π_t are adapted, these derived sets also change. For instance if a block in Π_s is replaced by two other blocks (this happens at Lines 1.17 and 1.25 of Algorithm 1), the corresponding block-bunch-slices and action-block-slices are split as well. For the sake of brevity, we omit the updating of these derived sets in the high-level description of the algorithm. We describe how these sets are maintained in Section 5.

When Π_t is changed, the invariant needs to be re-established by splitting blocks. To keep track of the blocks that still need to be split, we partition the block-bunch-slices $T_{B\to}$ into *stable*

and *unstable* block-bunch-slices. A block-bunch-slice is stable if we have ensured that it is not a splitter for any block in Π_s. Otherwise it is deemed unstable, and it needs to be checked whether it is stable, or whether some block B must be split. If a block-bunch-slice is unstable, it is stored in the *splitter list,* either as a *primary* or as a *secondary* splitter. Moreover, if a block-bunch-slice is unstable, we divide the transitions in this block-bunch-slice in *marked* and *non-marked* transitions. These markings are used to determine which bottom states in a block have a transition in a particular bunch, and are essential for efficient splitting of blocks. For an unstable block-bunch-slice $T_{B\to}$ we write its marked transitions as $\mathit{Marked}(T_{B\to})$. Note that, even when a block-bunch-slice $T_{B\to}$ resides on the splitter list, the block B may be split due to another splitter. In such a case, the block-bunch-slice is split accordingly, and the splitter list is implicitly adapted.

3.2 Overview of the algorithm

Before performing the partition refinement, the LTS is preprocessed to contract τ-strongly connected components (SCCs) into a single state without τ-loop. This step is valid since all states on a τ-SCC are branching bisimilar, and it ensures that all τ-paths in the LTS are finite.

The algorithm itself is a partition refinement algorithm. It iteratively refines the partitions Π_s and Π_t. The main objective for the algorithm is to guarantee the following: If for some block B', a block B has a transition in $B \xrightarrow{a} B'$ in an action-block-slice $T_{\xrightarrow{a}B'}$, then every bottom state in B has a transition in the same action-block-slice $T_{\xrightarrow{a}B'}$. Since there are no infinite τ-paths in the LTS, every state in B is guaranteed to inertly reach a bottom state, hence every state in B can inertly reach a transition in $T_{\xrightarrow{a}B'}$. Therefore, whenever this objective has been reached, every block is a branching bisimulation equivalence class.

To achieve this objective, the algorithm maintains the following weaker invariant, which keeps track of bunches instead of action-block-slices. If a block B has a transition in $B \xrightarrow{a} B'$ to some block B' in a bunch T, then every bottom state in B has a transition in T. Observe that, once every action-block-slice is in its own bunch, and this is reflected in the blocks, the main objective of the algorithm has been reached.

Hence, the main invariant of our algorithm is the following.

Invariant 3.2 (Bunches)**.** Π_s is stable under Π_t, i.e., if a bunch $T \in \Pi_t$ contains a transition with its source state in a block B of Π_s, then every bottom state in block B has a transition in bunch T (in fact, in block-bunch-slice $T_{B\to}$).

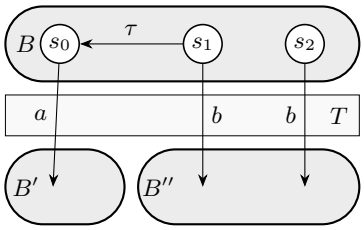
Now, if a bunch contains multiple action-block-slices—we call that a *non-trivial* bunch—, to get closer to our main objective, we have to refine Π_t by splitting off an action-block-slice. At the same time, to preserve the main invariant, when we split a bunch, we need to reflect this change in the blocks. Therefore, the blocks that had an inertly reachable transition in the original bunch are split into subblocks that can either inertly reach the new bunch, the remainder of the original bunch, or both. This idea is fleshed out in Algorithm 1. We first illustrate one step of the algorithm using the example in Figure 1.

Example 3.3. We start with a part of a labelled transition system in Figure 1a. We have three states, s_0, s_1, and s_2, with an inert transition $s_1 \xrightarrow{\tau} s_0$, and a number of transitions in the bunch T. Note that T is non-trivial: it contains two action-block-slices, $T_{\xrightarrow{a}B'}$ and $T_{\xrightarrow{b}B''}$.

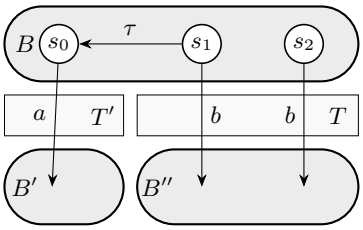
To get closer to the situation where every bunch consists of exactly one action-block slice, the algorithm splits off a small action-block-slice from bunch T, and puts it into its own bunch T'. This situation is shown in Figure 1b.

To ensure that the invariant is maintained, we now need to split block B with respect to T' and T. We first split with respect to T'. This splits block B into two blocks: R, which contains the states that can inertly reach a transition in T', and U, containing those states from which T' is unreachable inertly. This puts s_0 and s_1 in R, and s_2 in U. This is shown in Figure 1c.

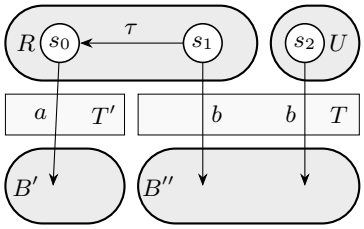
Block U is stable with respect to T since, according to the invariant that holds before splitting the bunch, all states that cannot inertly reach T' must be able to inertly reach T. Block R,



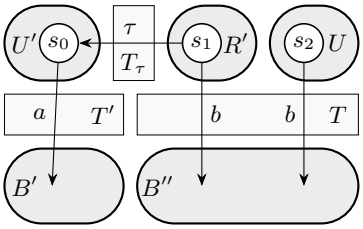
(a) Part of a labelled transition system with three states, s_0, s_1, s_2, divided into three blocks B, B' and B'', and a single non-trivial bunch T.



(b) Situation after moving a small action-block-slice $T_{\xrightarrow{a}B'}$ to its own bunch T'.



(c) Situation after splitting B with respect to T'.



(d) Situation after splitting R with respect to T.

Figure 1: One step in the branching bisimulation algorithm

however, is not yet stable with respect to T. Therefore, R must be split into the set of states from which a transition in T is unreachable, denoted U', and from which a transition in T is reachable, denoted R'. This is shown in Figure 1d.

Note that in this last split, the τ-transition becomes non-inert, and a new bunch T_τ containing this transition is introduced. The algorithm needs to stabilise with respect to this new bunch, and it needs to ensure that R', the block containing a new bottom state, is stabilised with respect to all other bunches it can inertly reach. In this particular example, R' happens to be stable with respect to all bunches, so in this case, the required stabilisation has no effect.

3.3 In-depth description of the algorithm

The pseudocode of the algorithm is given in Algorithm 1. The algorithm works as follows, where we start with the initialisation. First, at Line 1.1, all τ-SCCs are contracted into a single state each (without τ-loop). All states in a τ-SCC are branching bisimilar (as they can all reach the same states, possibly after a few inert transitions). This preprocessing ensures that there are no τ-cycles in the LTS (Invariant 3.7 below) and from every non-bottom state a bottom state can be reached via inert transitions (Lemma 3.8 below).

Second, at Lines 1.2–1.5, we create the initial partitions Π_s and Π_t as follows. Let B_{vis} be the set of states from which a visible transition is inertly reachable, and let B_{invis} be the other states (i.e., the states from which only a deadlock state can be inertly reached). Then $\Pi_s = \{B_{\text{vis}}, B_{\text{invis}}\} \setminus \{\emptyset\}$. Initially, Π_t contains one bunch consisting of all non-inert transitions. All the block-bunch-slices induced by Π_s and Π_t are initially stable, i.e., for all block-bunch-slices $T_{B\to}$, every bottom state of B, which must be B_{vis}, has a transition in $T_{B\to}$ (this is Invariant 3.2 above).

If there are non-trivial bunches, these bunches need to be split such that they ultimately become trivial. The outer loop of the algorithm (Lines 1.6–1.31) takes a non-trivial bunch T from Π_t, and from this it moves a *small* (containing at most half the number of transitions in the bunch) action-block-slice $T_{\xrightarrow{a}B'}$ into its own bunch in Π_t (Line 1.8). Hence, bunch T is reduced to $T \setminus T_{\xrightarrow{a}B'}$.

The two new bunches $T_{\xrightarrow{a}B'}$ and $T \setminus T_{\xrightarrow{a}B'}$ can cause instability, violating Invariant 3.2. This means there can be blocks with transitions in one new bunch, but not all bottom states have transitions in that bunch because some bottom states only have transitions in the other new bunch.

Algorithm 1 Abstract algorithm for branching bisimulation partitioning

1.1: Find τ-SCCs and contract each of them to a single state
1.2: $B_{\text{vis}} := \{s \in S \mid \exists s_0, \ldots, s_n, s' \in S, a \in Act \setminus \{\tau\}. s = s_0 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_n \xrightarrow{a} s'\}$
1.3: $B_{\text{invis}} := S \setminus B_{\text{vis}}$ } $O(m)$
1.4: $\Pi_s := \{B_{\text{vis}}, B_{\text{invis}}\} \setminus \{\emptyset\}$
1.5: $\Pi_t := \{\{s \xrightarrow{a} s' \mid a \in Act \setminus \{\tau\}, s, s' \in S\} \cup B_{\text{vis}} \xrightarrow{\tau} B_{\text{invis}}\}$
1.6: **while** a $T \in \Pi_t$ exists with $\#aB'(T) > 1$ **do** } $\leq m$ iterations
1.7: Select some $a \in Act$ and $B' \in \Pi_s$ such that $|T_{\rightarrow B'}^{\xrightarrow{a}}| \leq \frac{1}{2}|T|$
1.8: $\Pi_t := (\Pi_t \setminus \{T\}) \cup \{T_{\rightarrow B'}^{\xrightarrow{a}}, T \setminus T_{\rightarrow B'}^{\xrightarrow{a}}\}$
1.9: **for all** $B \in$ *splittableBlocks*$(T_{\rightarrow B'}^{\xrightarrow{a}})$ **do**
1.10: Add first $T_{B \rightarrow B'}^{\xrightarrow{a}}$ and then $T_{B\rightarrow} \setminus T_{B \rightarrow B'}^{\xrightarrow{a}}$ to the splitter list. Label $T_{B \rightarrow B'}^{\xrightarrow{a}}$ primary and $T_{B\rightarrow} \setminus T_{B \rightarrow B'}^{\xrightarrow{a}}$ secondary } $O(|T_{\rightarrow B'}^{\xrightarrow{a}}|)$
1.11: Mark all transitions in $T_{B \rightarrow B'}^{\xrightarrow{a}}$
1.12: For every state with both marked outgoing transitions and an outgoing transition in $T_{B\rightarrow} \setminus T_{B \rightarrow B'}^{\xrightarrow{a}}$, mark one such transition
1.13: **end for**
1.14: **for all** $T'_{B\rightarrow}$ in the splitter list (in order) **do** } $\leq |T_{\rightarrow B'}^{\xrightarrow{a}}|$ iterations
1.15: $\langle R, U\rangle := \mathsf{split}(B, T'_{B\rightarrow})$ // $R \supseteq B_{\xrightarrow{T'}}$ can reach $T'_{B\rightarrow}$ and // $U = B \setminus R$ cannot reach it
1.16: Remove $T'_{B\rightarrow} = T'_{R\rightarrow}$ from the splitter list
1.17: $\Pi_s := (\Pi_s \setminus \{B\}) \cup (\{R, U\} \setminus \{\emptyset\})$ } $O(|Marked(T'_{B\rightarrow})| + |U_{\rightarrow}| + |U_{\leftarrow}| + |Bottom(N)_{\rightarrow}|)$ or $O(|R_{\rightarrow}| + |R_{\leftarrow}|)$
1.18: **if** $T'_{B\rightarrow}$ is primary (note: $T'_{B\rightarrow} = T_{B \rightarrow B'}^{\xrightarrow{a}}$) **then**
1.19: Remove $T_{U\rightarrow} \setminus T_{U \rightarrow B'}^{\xrightarrow{a}}$ from the splitter list
1.20: **end if**
1.21: **if** $R \xrightarrow{\tau} U \neq \emptyset$ **then**
1.22: Create a new bunch containing exactly $R \xrightarrow{\tau} U$, add $R \xrightarrow{\tau} U = (R \xrightarrow{\tau} U)_{R\rightarrow}$ to the splitter list, and mark all its transitions
1.23: $\langle N, R'\rangle := \mathsf{split}(R, R \xrightarrow{\tau} U)$ // $N \supseteq R_{\xrightarrow{R \xrightarrow{\tau} U}}$ contains // the new bottom states } $O(|R \xrightarrow{\tau} U| + |R'_{\rightarrow}| + |R'_{\leftarrow}| + |Bottom(N)_{\rightarrow}|)$ or $O(|N_{\rightarrow}| + |N_{\leftarrow}|)$
1.24: Remove $R \xrightarrow{\tau} U = (R \xrightarrow{\tau} U)_{N\rightarrow}$ from the splitter list
1.25: $\Pi_s := (\Pi_s \setminus \{R\}) \cup (\{N, R'\} \setminus \{\emptyset\})$
1.26: Add $N \xrightarrow{\tau} R'$ to the bunch containing $R \xrightarrow{\tau} U$
1.27: Add all $T_{N\rightarrow}$ to the splitter list and label them secondary } $O(|Bottom^*(N)_{\rightarrow}|)$
1.28: For each bottom state, mark one of its outgoing transitions in every $T_{N\rightarrow}$ where it has one } $O(|Bottom(N)_{\rightarrow}|)$
1.29: **end if**
1.30: **end for**
1.31: **end while**

For such blocks, stability needs to be restored by splitting them. The set *splittableBlocks*$(T_{\rightarrow B'}^{\xrightarrow{a}})$ contains all blocks that have transitions in both new bunches $T_{\rightarrow B'}^{\xrightarrow{a}}$ and $T \setminus T_{\rightarrow B'}^{\xrightarrow{a}}$; these blocks must be investigated. All other blocks are stable with respect to the new bunches.

Earlier algorithms would now investigate all blocks to re-establish stability. Instead, we investigate all block-bunch-slices in the smaller of the two new bunches. All blocks that do not have transitions in these block-bunch-slices are stable with respect to both bunches. The first inner loop (Lines 1.9–1.13) serves to insert all these block-bunch-slices into the splitter list. Block-bunch-slices of the shape $T_{B \rightarrow B'}^{\xrightarrow{a}}$ in the splitter list are labelled primary, and all other block-bunch-slices in this list are labelled secondary.

In the first loop (Lines 1.9–1.13) some transitions are marked. The function of this marking is similar to that of the counters in [17]: it serves to determine quickly whether a bottom state has a transition in a secondary splitter $T_{B\rightarrow} \setminus T_{B \rightarrow B'}^{\xrightarrow{a}}$ (or slices that are the result of splitting this slice). Invariant 3.11 below indicates that a bottom state has transitions in some splitter block-bunch-slice if and only if it has marked transitions in this slice. The marked transitions are stored separately in a block-bunch-slice and therefore can be visited without spending time on unmarked transitions of the block. This is essential to obtain the time complexity of the algorithm, as we are

allowed to perform one unit of work per transition in $T_{\stackrel{a}{\to}B'}$ (the smaller of the two new bunches), and since $|\mathit{Marked}(T_{B\to} \setminus T_{B\stackrel{a}{\to}B'})| \leq |T_{B\stackrel{a}{\to}B'}|$, we do not mark too many transitions.

In the second loop (Lines 1.14–1.30), one splitter $T'_{B\to}$ from the splitter list is taken at a time and its source block is split into R (the part that can inertly reach some transition in $T'_{B\to}$) and U (the part that cannot inertly reach $T'_{B\to}$) to re-establish stability. Formally, the routine $\mathsf{split}(B, T)$ delivers the pair $\langle R, U\rangle$ defined by:

$$\begin{aligned} &R = \{s \in B \mid s \stackrel{\tau}{\to} s_1 \stackrel{\tau}{\to} \cdots \stackrel{\tau}{\to} s_n \stackrel{a}{\to} s' \text{ where } \{s_1, \ldots, s_n\} \subseteq B \text{ and } s_n \stackrel{a}{\to} s' \in T\},\\ &U = B \setminus R. \end{aligned} \tag{1}$$

We will detail its algorithm and argue for its correctness in Section 4.

If $T'_{B\to}$ was a primary splitter of the form $T_{B\stackrel{a}{\to}B'}$ added to the splitter list at Line 1.10, then we know that U must be stable under $T_{U\to} \setminus T_{U\stackrel{a}{\to}B'}$, as every bottom state in B has a transition in the former block-bunch-slice $T_{B\to}$, and as the states in U have no transition in $T_{B\stackrel{a}{\to}B'}$, every bottom state in U must have a transition in $T_{B\to} \setminus T_{B\stackrel{a}{\to}B'}$. Therefore, at Line 1.19, block-bunch-slice $T_{U\to} \setminus T_{U\stackrel{a}{\to}B'}$ can be removed from the splitter list without investigating it. This is crucial for the complexity, and essentially it is the translation of the three-way split from [17].

Some invisible transitions may have become non-inert, namely the τ-transitions that go from R to U. There cannot be τ-transitions from U to R as otherwise, from the source state of such a transition, $T'_{B\to}$ could be inertly reached, so it should have been added to R instead of U. The new non-inert transitions were not yet part of a bunch in Π_t. So, a new bunch $R \stackrel{\tau}{\to} U$ is formed containing these transitions. All transitions in this new bunch leave block R and therefore R is the only block that may not be stable under this new bunch. To re-establish stability we split R into blocks $\langle N, R'\rangle$ (Line 1.23). Observe that there can be transitions $N \stackrel{\tau}{\to} R'$ that also become non-inert, and we add those to the new bunch $R \stackrel{\tau}{\to} U$.

In N, i.e., the set of states that can inertly reach some transition in $R \stackrel{\tau}{\to} U$, some states are new bottom states, while R' contains all original bottom states of R. In accordance with the observations in [10] blocks containing new bottom states can become unstable under any block-bunch-slice. Therefore, stability under all those block-bunch-slices must be re-established and therefore all the block-bunch-slices leaving block N are put on the splitter list. We mark one of the transitions in every new bottom state such that we can find the bottom states with a transition in $T_{N\to}$ in time proportional to the number of such new bottom states.

This algorithm is repeated until all action-block-slices coincide with the bunches. In the next section we prove that the resulting partition Π_s exactly coincides with branching bisimilarity.

We illustrate the algorithm in the following example. Note that the example also illustrates some of the details of the split subroutine, which is discussed in detail in Section 4.

Example 3.4. Consider the situation in Figure 2a. Observe that block B is stable w.r.t. the bunches T and T'. We have split off a small bunch $T_{\stackrel{a}{\to}B'}$ from T, and as a consequence, B needs to be restabilised. The bunches put on the splitter list initially are $T_{\stackrel{a}{\to}B'}$ and $T \setminus T_{\stackrel{a}{\to}B'}$. When putting these bunches on the splitter list, all transitions in $T_{B\stackrel{a}{\to}B'}$ are marked, see the m's in Figure 2b. Also, for states that have transitions both in $T_{\stackrel{a}{\to}B'}$ and in $T \setminus T_{\stackrel{a}{\to}B'}$, one such transition in the latter bunch is marked, see the m's in Figure 2b.

We now first split B w.r.t. the primary splitter $T_{\stackrel{a}{\to}B'}$ into R, the states that can inertly reach $T_{\stackrel{a}{\to}B'}$, and U, the states that cannot. In Figure 2b, the states known to be destined for R are indicated ◐, the states known to be destined for U are indicated ⊖. Initially, all states with a marked outgoing transition are destined for R, the remaining bottom state of B is destined for U. The split subroutine proceeds to extend sets R and U in a backwards fashion using two coroutines, marking a state destined for R if one of its successors is already in R, and marking a state destined for U if all its successors are in U. In this case, the state in U does not have any incoming inert transitions, so its coroutine immediately terminates and all other states belong to R. Block B is split into R and U. The resulting block U is stable w.r.t. both $T_{\stackrel{a}{\to}B'}$ and $T \setminus T_{\stackrel{a}{\to}B'}$. The resulting sets R and U are shown in Figure 2c.

We still need to split R w.r.t. $T \setminus T_{\stackrel{a}{\to}B'}$, into R_1 and U_1, say. For this, we use the marked transitions in $T \setminus T_{\stackrel{a}{\to}B'}$ as a starting point to compute all bottom states that can reach a transition

in $T \setminus T_{\xrightarrow{a}B'}$. This guarantees that the time we use is proportional to the size of $T_{\xrightarrow{a}B'}$. Initially, there is one state destined for R_1, marked ◍ in Figure 2c, and one state destined for U_1, marked ⊖ in the same figure. We now perform both coroutines in split simultaneously. Figure 2d shows the situation after both coroutines have considered one transition: The U_1-coroutine (the coroutine that calculates the states that cannot inertly reach a transition in $T \setminus T_{\xrightarrow{a}B'}$) has initialised the counter *untested* of one state to 2 on Line 2.10 of Algorithm 2 because two of its outgoing inert transitions have not yet been considered. The R_1-coroutine (the coroutine that calculates the states that can inertly reach a transition in $T \setminus T_{\xrightarrow{a}B'}$) has checked the unmarked transition in the splitter $T_{R\to} \setminus T_{R\xrightarrow{a}B'}$. As the latter coroutine has finished visiting unmarked transitions in the splitter, the U_1-coroutine will no longer need to run the slow test loop at Lines 2.14–2.18 of the left-most column of Algorithm 2. In Figure 2e the situation is shown after two more steps in the coroutines. Each has visited two extra transitions. There are two extra states destined for R_1, marked ◍, and there is one state destined for U_1 with 0 remaining inert transitions, for which we know immediately that it has no transition in $T \setminus T_{\xrightarrow{a}B'}$, this is marked ⊖. Now, the R_1-coroutine is terminated, since it contains more that $\frac{1}{2}|R|$ states, and the remaining incoming transitions of states in U_1 are visited. This will not further extend U_1 since all states are already either destined for R_1 or for U_1.

The result of splitting is shown in Figure 2f. Note that some inert transitions become non-inert, so a new bunch with the transitions $R_1 \xrightarrow{\tau} U_1$ is created, and all those transitions are marked m.

We next have to split R_1 with respect to this new bunch into the set of states N_1 that can inertly reach a transition in the new bunch, and the set R'_1 that cannot inertly reach this bunch. In this case, all states in R_1 have a marked outgoing transition, hence $N_1 = R_1$, and $R'_1 = \emptyset$. The coroutine that calculates the set of states that cannot inertly reach a transition in the bunch will immediately terminate because there are no transitions to be considered.

Observe that R_1 ($= N_1$) has a new bottom state. This means that stability of R_1 with respect to any bunch is not guaranteed any more, and this stability needs to be re-established. We therefore consider all bunches in which R_1 has an outgoing transition. We add $T_{R_1\xrightarrow{a}B'}$, $T_{R_1\to} \setminus T_{R_1\xrightarrow{a}B'}$ and $T'_{R_1\to}$ to the splitter list as secondary splitters, and mark one of the outgoing transitions from each bottom state in each of those bunches using m. This situation is shown in Figure 2g.

In this case, R_1 is stable w.r.t. $T_{R_1\xrightarrow{a}B'}$ and $T_{R_1\to} \setminus T_{R_1\xrightarrow{a}B'}$, i.e., all states in R_1 can inertly reach a transition in both bunches. In both cases this is observed immediately after initialisation in split, since the set of states that cannot inertly reach a transition in these bunches is initially empty, and the corresponding coroutine terminates immediately.

Therefore, consider splitting R_1 with respect to $T'_{R_1\to}$. This leads to R_2, the set of states that can inertly reach a transition in T', and U_2, the set of states that cannot inertly reach a transition in T'. Note there are no marked transitions in $T'_{R_1\to}$, so initially all bottom states of R_1 are destined for U_2 (marked ⊖ in Figure 2h), and there are no states destined for R_2. Then we start splitting R_1. In the R_2-coroutine, we first add the states with an unmarked transition in $T'_{R_1\to}$ to R_2 at Line 2.5r (i.e., in the right column of Algorithm 2) and then all predecessors of the new bottom state need to be considered. When split terminates, there will be no additional states in U_2, and the remaining states end up in R_2.

The situation after splitting R_1 into R_2 and U_2 is shown in Figure 2i. One of the inert transitions becomes non-inert, this is marked m. Furthermore, R_2 contains a new bottom state. This is the state in R_2 with a transition to T', and it is marked nb. Note that, as each block necessarily has a bottom state (see Lemma 3.8), a non-bottom state had to become a bottom state in this case.

We need to continue stabilising R_2 w.r.t. the bunch $R_2 \xrightarrow{\tau} U_2$, which does not lead to a new split, and we need to restabilise R_2 w.r.t. all bunches in which it has an outgoing transition. This also does not lead to new splits, so the situation in Figure 2i after removing the marking of transitions in $R_2 \xrightarrow{\tau} U_2$ is the final result of splitting.

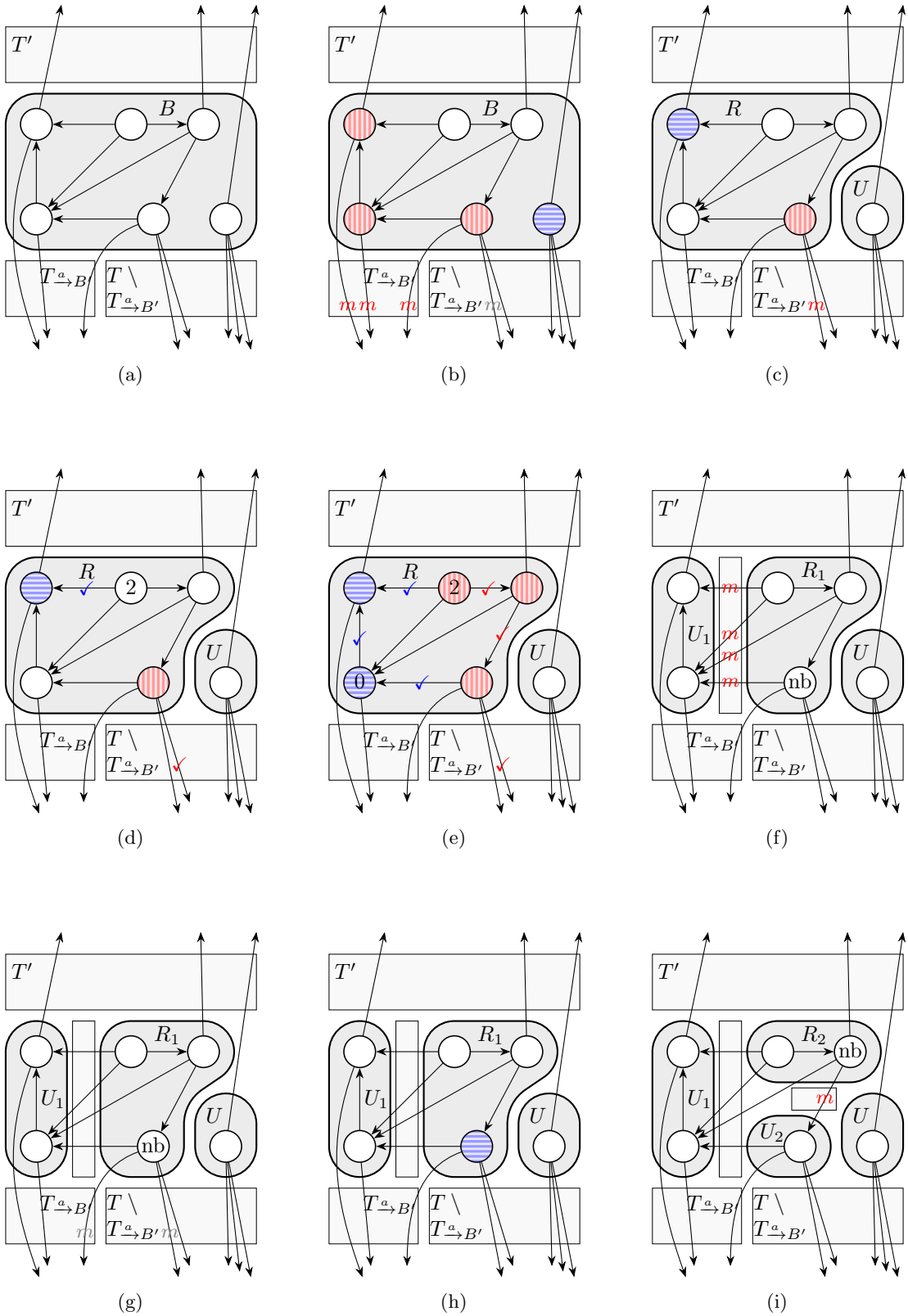


Figure 2: Illustration of splitting of a small block from T and stabilising block B with respect to the new bunches $T_{\xrightarrow{a}B'}$ and $T \setminus T_{\xrightarrow{a}B'}$, as explained in Example 3.4.

3.4 Correctness

The validity of the algorithm follows from a number of major invariants.

The main invariant is Invariant 3.2, which was stated before. The following invariant indicates that two non-inert transitions starting in the same block, having the same label and ending in the same block, will always reside in the same bunch.

Invariant 3.5 (Bunches are not unnecessarily split)**.** For any pair of non-inert transitions $s \xrightarrow{a} s'$ and $t \xrightarrow{a} t'$, if $s, t \in B$ and $s', t' \in B'$ then $s \xrightarrow{a} s' \in T$ and $t \xrightarrow{a} t' \in T$ for some bunch $T \in \Pi_t$.

Proof. Initially, Π_t contains one bunch with all non-inert transitions. Therefore, the invariant is valid.

There are two places in the algorithm where validity of this invariant can be jeopardised. At Line 1.8 Π_t is refined, and at Lines 1.22 and 1.26 a new bunch is created and extended.

We first look at Line 1.8. We replace bunch T with the two bunches $T_{\xrightarrow{a}B'}$ and $T \setminus T_{\xrightarrow{a}B'}$. As all the transitions in $T_{\xrightarrow{a}B'}$ have label a and go to block B', the invariant is maintained for $T_{\xrightarrow{a}B'}$. All transitions in $T \setminus T_{\xrightarrow{a}B'}$ have a label different from a or go to a different block than B', so the invariant is maintained for these transitions as well.

At Lines 1.22 and 1.26 all new non-inert transitions are put in a new bunch. As these are the only τ-transitions between blocks R and U, the invariant remains valid under the creation of this bunch. □

The following invariant says that states that are branching bisimilar will never end up in separate blocks.

Invariant 3.6 (Preservation of branching bisimilarity)**.** For all states $s, t \in S$, if $s \leftrightarrows_b t$, then there is some block $B \in \Pi_s$ such that $s, t \in B$.

Proof. Initially, this invariant is valid. We prove this by contraposition. Consider two states s, t in different blocks. Then $s \in B_{\text{vis}}$ and $t \in B_{\text{invis}}$, or vice versa. But then s and t cannot be branching bisimilar, because from s a visible action is inertly reachable, whereas that is not the case from t (or vice versa).

The preservation of this invariant can be seen as follows. There are two places where blocks in Π_s are split, namely at Lines 1.17 and 1.25. We first concentrate on the splitting at Line 1.17. In this case a block B is split into R and U where $\langle R, U\rangle$ is the result of an invocation of $\mathsf{split}(B, T'_{B\to})$. Assume that the invariant would be invalidated. This means that there are states $s \in R$ and $t \in U$ with $s \leftrightarrows_b t$. As $s \in R$, by Equation (1) on page 9 we know that $s \xrightarrow{\tau} s_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_n \xrightarrow{a} s'$ with $\{s_1, \ldots, s_n\} \subseteq B$, $s_n \xrightarrow{a} s' \in T'_{B\to}$ and $s' \in B'$. As $s_n \xrightarrow{a} s'$ is non-inert and $s \leftrightarrows_b t$, we must have $t \xrightarrow{\tau} t_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} t_m \xrightarrow{a} t'$, $t_j \in B$ for all $1 \le j \le m$ and $t' \in B'$. Hence, by Invariant 3.5, $t_m \xrightarrow{a} t' \in T'$ where T' is the bunch such that $T'_{B\to} \subseteq T'$. But as $T'_{B\to}$ is a block-bunch-slice, $t_m \xrightarrow{a} t' \in T'_{B\to}$. Hence, by (1) on page 9 it follows that $t \in R$, contradicting the assumption that the invariant could be invalidated.

At Line 1.25 splitting takes place with regard to a splitter $R \xrightarrow{\tau} U$. As this is a bunch satisfying Invariant 3.5, the reasoning is completely analogous to the previous case. □

The following invariant says that there are no τ-loops in any block. Actually, a stronger property holds, namely that there are no τ-loops at all, after they have been removed during the initialisation of the algorithm.

Invariant 3.7 (No inert loops)**.** There is no inert loop in a block, i.e., for every sequence $s_1 \xrightarrow{\tau} s_2 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_n$ with $s_i \in B \in \Pi_s$ it holds that $s_i \neq s_j$ for all $1 \le i < j \le n$.

Proof. Initially, this invariant holds because every strongly connected component consisting of states connected via τ-transitions is contracted and merged into a single state. The only operation that influences this invariant is splitting a block (Lines 1.17 and 1.25). Splitting blocks cannot introduce new loops. □

As a consequence of the previous invariant, every block has at least one bottom state, and from every state in a block a bottom state can be inertly reached.

Lemma 3.8. *Invariant 3.7 implies that for all partitions Π_s of S, and all blocks B in Π_s, we have*

1. *$Bottom(B) \neq \emptyset$.*
2. *For every state $s \in B$, there is a path of inert transitions leading to a bottom state in B.*

Proof. Let Π_s be an arbitrary partition of S, and $B \in \Pi_s$ be a block in S such that $B \neq \emptyset$.

1. Towards a contradiction, assume that $Bottom(B) = \emptyset$. Then every state s in B has an outgoing transition $s \xrightarrow{\tau} s'$ for some $s' \in B$. Since B is finite, there must be a τ-cycle in B. This contradicts Invariant 3.7.
2. Let $s \in B$ be arbitrary, and assume that s does not have a path of inert transitions leading to a bottom state in B. Then, there must, again, be a τ-cycle in B, which contradicts Invariant 3.7. □

Invariant 3.9 is a technical invariant required to prove the main Invariant 3.2. Invariant 3.9 holds for the second inner for loop, and says that Invariant 3.2 holds, except for the block-bunch-slices on the splitter list, for which the invariant has to be re-established by splitting blocks.

Invariant 3.9 (Inner loop at Lines 1.14–1.30)**.** If a non-empty block-bunch-slice $T_{B\to}$ is not in the splitter list, then every bottom state in its source block B has a transition in $T_{B\to}$.

Proof. First it is shown that the invariant holds when arriving at the second for loop at Line 1.14. Consider a non-empty block-bunch-slice $\hat{T}_{\hat{B}\to}$ that is not in the splitter list. If $\hat{T}_{\hat{B}\to}$ is not a subset of T at Line 1.6, it follows from Invariant 3.2 that all $t \in Bottom(\hat{B})$ have a transition in $\hat{T}$, so the invariant holds.

Now assume $\hat{T}_{\hat{B}\to}$ is a subset of T. Note that $\hat{T} = T$. Then one of the following cases apply:

- $\hat{T}_{\hat{B}\to} = T_{\hat{B}\xrightarrow{a}B'} \subseteq T_{\xrightarrow{a}B'}$. As $T_{\hat{B}\xrightarrow{a}B'}$ is not empty, there is a transition $s \xrightarrow{a} s' \in T_{\hat{B}\xrightarrow{a}B'} \subseteq T$. Towards a contradiction, suppose there is some state $t \in Bottom(\hat{B})$ that does not have an outgoing transition in $T_{\xrightarrow{a}B'}$. Then, t must have a transition in $T \setminus T_{\xrightarrow{a}B'}$ by Invariant 3.2, thus $\hat{B} \in splittableBlocks(T_{\xrightarrow{a}B'})$ and hence $T_{\hat{B}\xrightarrow{a}B'}$ is on the splitter list. This contradicts the assumption that $\hat{T}_{\hat{B}\to}$ is not on the splitter list.
- $\hat{T}_{\hat{B}\to} = T_{\hat{B}\to} \subseteq T \setminus T_{\xrightarrow{a}B'}$. It cannot happen that $\hat{B} \in splittableBlocks(T_{\xrightarrow{a}B'})$, since then $T_{\hat{B}\to}$ is on the splitter list. Therefore, $\hat{B} \notin splittableBlocks(T_{\xrightarrow{a}B'})$. Since $T_{\hat{B}\to} \neq \emptyset$, we have $T_{\hat{B}\to} = T$, and it immediately follows from Invariant 3.2 that all $t \in Bottom(\hat{B})$ have a transition in $T_{\hat{B}\to}$.

We now show that the loop starting at Line 1.14 maintains this invariant. Concretely, we consider some block-bunch-slice $\hat{T}_{\hat{B}\to}$ that is not empty and does not occur in the splitter list at Line 1.30. We make a case distinction on the block $\hat{B}$.

- Let us first assume that $\hat{B}$ is not the result of splitting a block at Lines 1.17 or 1.25. So, $\hat{B} \cap B = \emptyset$. This means that $\hat{T}_{\hat{B}\to}$ was not split during the last iteration of the for loop, and it is not a subset of the bunch with new non-inert transitions created at Lines 1.22 and 1.26. Hence, the invariant remains valid for $\hat{T}_{\hat{B}\to}$ during this last iteration.
- Assume $\hat{B}$ is a subset of R and the condition at Line 1.21 is not valid, i.e., $R \xrightarrow{\tau} U = \emptyset$. This means $\hat{B}$ is not split further at Line 1.25, i.e., $\hat{B} = R$. We treat the cases where $\hat{T}_{R\to}$ is or is not a subset of $T'_{B\to}$ separately.
  - If $\hat{T}_{R\to} \subseteq T'_{B\to}$ then we see that due to the splitting at Line 1.17 all states in R can reach a transition in $T'_{B\to} = T'_{R\to}$. In particular, all bottom states of $\hat{B}$ have a transition in $T'_{B\to}$ and this is also the case for $\hat{T}_{R\to}$ by construction. Hence, the invariant is valid for $\hat{T}_{R\to}$.

– $\hat{T}_{R\to} \cap T'_{B\to} = \emptyset$. In this case, $\hat{T}_{R\to}$ is the result of splitting a block-bunch-slice $\hat{T}_{B\to}$ that was not on the splitter list, as splitting $\hat{T}_{B\to}$ results in keeping the splitted block-bunch-slices on this list in case the original was already there. This means that all bottom states in B have transitions in $\hat{T}_{B\to}$ by Invariant 3.2, and by construction, those bottom states in R have transitions in $\hat{T}_{R\to}$. As $R \xrightarrow{\tau} U = \emptyset$, there are no additional bottom states in $\mathit{Bottom}(R) \setminus \mathit{Bottom}(B)$ and the invariant is established for $\hat{T}_{R\to}$.

- Assume $\hat{B}$ is a subset of R and the condition at Line 1.21 is valid. This means that $\hat{B}$ is either equal to N or R'.

  – Assume $\hat{B}$ equals N. This means that $\hat{T}_{\hat{B}\to}$ has the shape $T_{N\to}$ (Line 1.27) and is added to the splitter list, contradicting our assumption that $\hat{T}_{\hat{B}\to}$ is not on the splitter list at Line 1.30.

  – Assume $\hat{B}$ is equal to R'. In this case the block-bunch-slice $\hat{T}_{R'\to}$ cannot be a subset of the new bunch created at Lines 1.22 and 1.26 as there are no transitions in $(R \xrightarrow{\tau} U) \cup (N \xrightarrow{\tau} R')$ from R'.

    If the original block-bunch-slice $\hat{T}_{B\to}$ that existed at the beginning of the second for loop satisfying $\hat{T}_{R'\to} \subseteq \hat{T}_{B\to}$ was in the splitter list, then also $\hat{T}_{R'\to}$ would be on the list. Contradiction.

    Therefore, $\hat{T}_{B\to}$ was not in the splitter list. As $\hat{T}_{B\to}$ was not empty, all bottom states in B would have transitions in $\hat{T}_{B\to}$. As all new bottom states are moved to N, all bottom states in R' are also bottom states in B, and it follows that all bottom states in R' have transitions in $\hat{T}_{R'\to}$. Hence, the invariant holds.

- Here we consider the situation where $\hat{B} = U$. There are three cases to consider.

  – If $\hat{T}_{U\to} = T'_{U\to}$, we see that $T'_{U\to}$ is empty, because it contains all transitions in $T'_{B\to}$ reachable from states in U, from which, by construction, transitions in $\hat{T}_{B\to}$ cannot be reached. As $T'_{U\to}$ is empty, the invariant holds trivially.

  – Suppose $\hat{T}_{U\to} = T_{U\to} \setminus T_{U \xrightarrow{a} B'}$ and $T'_{B\to} = T_{B \xrightarrow{a} B'}$ is primary. We know that B is stable w.r.t. $T_{B\to}$ by Invariant 3.2. That means that every bottom state of B has a transition in $T_{B\to}$. Bottom states in U have no transitions in $T_{\xrightarrow{a} B'}$. Hence, they all have transitions in $T \setminus T_{\xrightarrow{a} B'}$. If we restrict this set of transitions to those starting in U, we see that all bottom states in U also have a transition in $T_{U\to} \setminus T_{U \xrightarrow{a} B'}$ and the invariant holds.

  – In this last case we investigate the remaining situations. So either $\hat{T}_{U\to} \neq T'_{U\to}$, or $\hat{T}_{U\to} \neq T_{U\to} \setminus T_{U \xrightarrow{a} B'}$ or $T'_{B\to}$ is not primary.

    If the original block-bunch-slice $\hat{T}_{B\to}$ that existed at the beginning of the second for loop satisfying $\hat{T}_{U\to} \subseteq \hat{T}_{B\to}$ was in the splitter list, then also $\hat{T}_{U\to}$ would be on the list. Contradiction.

    Therefore, $\hat{T}_{B\to}$ was not in the splitter list. As $\hat{T}_{U\to}$ is not empty, $\hat{T}_{B\to}$ is also not empty and so, all bottom states in B have a transition in $\hat{T}_{B\to}$. All bottom states of U are bottom states from B. (Otherwise there must have been a transition $s \xrightarrow{\tau} s'$ from a state $s \in U$ to a state $s' \in R$; but then s would be part of R.) Hence, all bottom states in U have a transition in $\hat{T}_{U\to}$, showing that the invariant holds. □

Invariant 3.2 is the main invariant of the algorithm. It is valid at Line 1.6. It is an adaptation to branching bisimulation of a similar invariant in [19]. It says that the partition is always stable w.r.t. the bunches. Stability refers to the presence of a transition in a bunch, and hence does not relate to actions or target blocks. We finally prove its invariance. The proof relies on Invariant 3.9.

Proof of Invariant 3.2. We show that Π_s is stable under Π_t, i.e., if a bunch $T \in \Pi_t$ contains a transition with its source state in a block B of Π_s, then every bottom state in block B has a transition in bunch T (in fact, in block-bunch-slice $T_{B\to}$).

Initially, this invariant is valid: The initial blocks in Π_s are B_{vis} and B_{invis}. Furthermore, there is only one bunch. From the states in B_{invis} no visible transition is reachable, and this means that all transitions from states in B_{invis} are inert. Therefore, they do not occur in the initial bunch and Invariant 3.2 holds trivially. Each bottom state in B_{vis} has an outgoing visible transition, which must occur in the initial bunch. This also makes the invariant valid in a trivial way, albeit for a different reason.

The invariant is invalidated when Π_t is split at Line 1.8. At the end of the second for loop (Line 1.30), emptiness of the splitter list and Invariant 3.9 imply that all block-bunch-slices are stable. This implies the invariant as follows. Consider a bunch $T \in \Pi_t$. Assume there is a transition $s \xrightarrow{a} s' \in T$ with $s \in B$. The transition $s \xrightarrow{a} s'$ occurs in the block-bunch-slice $T_{B\to}$. As $T_{B\to}$ is stable, it holds that every state $t \in \mathit{Bottom}(B)$ has an outgoing transition in $T_{B\to}$ and therefore the invariant holds at Line 1.30. □

The invariants given above allow us to prove that the algorithm works correctly. When the algorithm terminates (and this always happens, see Section 3.5), branching bisimilar states are perfectly grouped in blocks.

Theorem 3.10. *From the Invariants 3.6, 3.7 and 3.2, after the algorithm terminates, it holds that* $\equiv_{\Pi_s} = \leftrightarrows_b$.

Proof. By Invariant 3.6 it follows that $\leftrightarrows_b \subseteq \equiv_{\Pi_s}$. Hence, we only need to show that $\equiv_{\Pi_s} \subseteq \leftrightarrows_b$. We do this by showing that $\equiv_{\Pi_s}$ is a branching bisimulation relation.

Consider states $s, t \in S$ such that $s, t \in B$ for some block $B \in \Pi_s$. Assume $s \xrightarrow{a} s'$.

- If $a = \tau$ and $s' \in B$, i.e., the transition is inert, then $s' \equiv_{\Pi_s} t$ and we have fulfilled the proof obligation for branching bisimulation in this case.
- If the transition $s \xrightarrow{a} s'$ is not inert, it is part of some bunch $T \in \Pi_t$. By Invariant 3.7 and Lemma 3.8 there is a path of inert transitions $t \xrightarrow{\tau} \cdots \xrightarrow{\tau} t'$ with $t' \in \mathit{Bottom}(B)$, and by Invariant 3.2 there is a transition $t' \xrightarrow{b} t'' \in T$. As the algorithm terminated, the condition at Line 1.6 is false, which means that $\#aB'(T) = 1$. In other words, T is equal to some action-block-slice. This must be $T_{\xrightarrow{a}B'}$, as $s \xrightarrow{a} s' \in T$ (where $s' \in B'$). Hence, $b = a$ and $t'' \in B'$, and the proof obligation for branching bisimulation has been fullfilled.

Concluding, $\equiv_{\Pi_s}$ is a branching bisimulation, and therefore $\equiv_{\Pi_s} = \leftrightarrows_b$. □

We provide the following invariant as a precondition for splitting blocks in Section 4. It says that whenever $\mathsf{split}(\hat{B}, \hat{T}_{\hat{B}\to})$ is called, the bottom states in block $\hat{B}$ with transitions in bunch $\hat{T}$ can be found by only looking at the marked transitions. Checking whether a state has a marked outgoing transition can be done in constant time, which is essential for the algorithm.

Invariant 3.11 (Marked transitions)**.** Whenever $\mathsf{split}(\hat{B}, \hat{T}_{\hat{B}\to})$ is called, it holds that

$$\mathit{Bottom}(\hat{B}) \xrightarrow{\hat{T}_{\hat{B}\to}} = \mathit{Bottom}(\hat{B}) \xrightarrow{\mathit{Marked}(\hat{T}_{\hat{B}\to})}.$$

Proof. Whenever $\mathsf{split}(\hat{B}, \hat{T}_{\hat{B}\to})$ is called, the block-bunch-slice $\hat{T}_{\hat{B}\to}$ is in the splitter list. There are four cases when a block-bunch-slice is inserted into the splitter list:

- $\hat{T}_{\hat{B}\to} = T_{\hat{B}\xrightarrow{a}B'}$ is a primary splitter of $\hat{B}$. Then, all transitions in $T_{\hat{B}\xrightarrow{a}B'}$ are marked at Line 1.11, so the invariant obviously holds.
- $\hat{T}_{\hat{B}\to}$ is a secondary splitter of $\hat{B}$ that has been added at Line 1.10. Note that for every block, we first split under its primary splitter $T_{\hat{B}\xrightarrow{a}B'}$ and then apply what remains of the secondary splitter only to R (or only to R' in case $R \xrightarrow{\tau} U$ is not empty, but for brevity, we only mention R below). So, the call is $\mathsf{split}(R, T_{R\to} \setminus T_{R\xrightarrow{a}B'})$. As every bottom state of R has a transition in $T_{\hat{B}\to}$, Line 1.12 ensures that every bottom state in R which already was a bottom state in $\hat{B}$ and which has a transition in $T_{R\to} \setminus T_{R\xrightarrow{a}B'}$, also has a marked transition in it. So, the invariant holds.

- $\hat{T}_{\hat{B}\to} = R \xrightarrow{\tau} U$ at Line 1.23. All transitions of $R \xrightarrow{\tau} U$ are marked at Line 1.22, so the invariant holds.
- $\hat{T}_{\hat{B}\to} = \hat{T}_{N\to}$ is a splitter of N added at Line 1.27. In that case, Line 1.28 ensures that every bottom state with a transition in $\hat{T}_{N\to}$ has a marked transition in it. Hence, also in this last case, the invariant holds.

Additionally, it can happen that a splitter itself is split. Then, the two new subslices keep their markings. If new bottom states result from the split, they are handled in a similar way to the last case above: in all relevant slices, each new bottom state with an outgoing transition in that slice also has a marked outgoing transition in that slice. □

3.5 Complexity

We show that the algorithm runs in time $O(m \log n)$, using the time budgets printed in grey at the right-hand side of the pseudocode, which indicate how much time each piece of code is allowed to spend. In Section 5 it is explained how the data structures meet these time budgets.

The initialisation (Lines 1.1–1.6) can be performed in $O(m)$ time, where for the calculation of the sets of states, the assumption that $n \leq m + 1$ is used. The calculation of the time complexity of the while loop is split into three separate parts. The first part regards splitting bunches, putting block-bunch-slices on the splitter list and marking transitions, which is attributed to the transitions in a new small bunch. The second part deals with splitting blocks, which is attributed to the transitions of the smaller subblock. The third part handles the calculations that are required when states become new bottom states.

When splitting bunches, we apply the "Process the smaller half" principle to new bunches. Every transition is an element of a new bunch $T_{\xrightarrow{a}B'}$ at Line 1.10 at most $\lfloor \log_2 n^2 \rfloor + 1$ times, because the first time $T_{\xrightarrow{a}B'}$ is investigated, it has at most n^2 elements, and every subsequent time a new bunch containing the same transition is investigated, it has at most half the size of the previous bunch. Whenever a transition is an element of a new bunch $T_{\xrightarrow{a}B'}$, it is processed in constant time. This is indicated with the time budget $O(|T_{\xrightarrow{a}B'}|)$ for Lines 1.7–1.14. Also, processing block-bunch-slices of the form $T_{B\xrightarrow{a}B'}$ and $T_{B\to} \setminus T_{B\xrightarrow{a}B'}$ (as added to the splitter list at Line 1.10) requires time in $O\big(\big|\mathit{Marked}(T'_{B\to})\big|\big)$ at Lines 1.15–1.22, corresponding to at most the number of transitions in $T_{B\xrightarrow{a}B'}$. Summing over all transitions gives runtime $O(m \log n)$ attributed to new bunches.

In the next section we explain in detail how long split can take. In short, its runtime depends on the *smaller* of the two resulting subblocks—so we apply "Process the smaller half" to that block. Every state can be part of such a smaller subblock at most $\lfloor \log_2 n \rfloor$ times, because the first time it is part of a subblock of at most $n/2$ states, and at every subsequent time the same state becomes part of another smaller subblock, the latter has at most half the size of the previous subblock. Whenever a state is in the smaller subblock of split, we are allowed to attribute time proportional to the number of incoming and outgoing transitions of that state. More precisely, provided each source state of T is in B, the complexity of calculating $\mathsf{split}(B, T)$ is the following. If $|R| \leq |U|$, then the time spent is $O(|R_\to| + |R_\leftarrow|)$, and if $|U| \leq |R|$, the time spent is $O(|\mathit{Marked}(T)| + |U_\to| + |U_\leftarrow| + |(\mathit{Bottom}(R) \setminus \mathit{Bottom}(B))_\to|)$.

In Algorithm 1 this is indicated with the time budget "$O(|U_\to| + |U_\leftarrow|)$ or $O(|R_\to| + |R_\leftarrow|)$" for Lines 1.15–1.22 and with "$O\big(\big|R'_\to\big| + \big|R'_\leftarrow\big|\big)$ or $O(|N_\to| + |N_\leftarrow|)$" for Lines 1.23–1.26. Also, as $R \xrightarrow{\tau} U \subseteq U_\leftarrow \cap R_\to$, we attribute $O(|R \xrightarrow{\tau} U|)$ at Lines 1.23–1.26 to U or R, whichever is smaller. Summing over all states gives rise to a cumulative time complexity of $O(m \log n)$.

Finally, some work is attributed to new bottom states. Every non-bottom state can become a bottom state at most once during the whole execution. When this happens, we attribute time proportional to its outgoing transitions to it. This is indicated with several time budgets $O(|\mathit{Bottom}(N)_\to|)$ at Lines 1.15–1.28. At Line 1.27 we need to include not only the current new bottom states but also the future ones because there may be block-bunch-slices that only have transitions from non-bottom states. When N is split under such a block-bunch-slice, at least one of these non-bottom states will become a bottom state but we cannot yet say which one(s) right

now. Also, for block-bunch-slices of the form $T_{N\to}$ at Line 1.28, we budget $O\big(\big|\mathit{Marked}(T'_{B\to})\big|\big)$ at Lines 1.15–1.22 corresponding to the new bottom states in N. Summing over all states gives runtime $O(m)$ attributed to new bottom states.

Adding up these three time budgets shows that the grand total of all work is $O(m \log n)$.

4 Splitting blocks

The function $\mathsf{split}(B, T)$ refines the block B into two subblocks, R and U, where R contains those states in B that can inertly reach a transition in T, and U contains the states that cannot, as formally specified in Equation (1).

These two sets are computed by two coroutines that are executed in such a way that the work between the two is balanced by alternating between the coroutines, and all work done in both coroutines can be attributed to the smaller of the two blocks R and U. Algorithm 2 presents those coroutines.

As a precondition, guaranteed by Invariant 3.11, the function requires that the marking of transitions in $T_{B\to}$ is such that bottom states of B that have an outgoing transition in $T_{B\to}$ also have a marked outgoing transition in $T_{B\to}$. Formally,

$$\mathit{Bottom}(B) \xrightarrow{\mathit{Marked}(T_{B\to})} = \mathit{Bottom}(B) \xrightarrow{T_{B\to}}.$$

The initial sets are computed as follows. Initially, all states in $B \xrightarrow{\mathit{Marked}(T)}$, i.e., all states that are the source of a marked transition in T, are put in R. All bottom states that are not in R initially are put in U. Observe that these initial sets can be computed in $O(|\mathit{Marked}(T)|)$ time by grouping bottom states with marked transitions.

The sets are extended as follows in the coroutines. For the states in R, first the states in $B \xrightarrow{T \setminus \mathit{Marked}(T)}$ are added that were not yet in R, i.e., all states that are the source of an unmarked transition in T, are added to R. Using backward reachability along inert transitions, R is extended until either R is stable (no states can be added), or R contains more than half the states in B.

To identify the states in U, observe that a state t is in U if either it is a bottom state that does not have a transition in T, or all successors of t are forced to inertly reach such a bottom state. To compute U, we keep track of a counter $\mathit{untested}[t]$ for every state t, that records the number of outgoing inert transitions to states that are not yet known to be in U. If $\mathit{untested}[t] = 0$, this means all inert successors of t are guaranteed to be in U, so, provided t does not have a transition in $T_{B\to}$, t is also added to U. To take care of the possibility that all inert transitions of t have been visited before all states that are the source of a transition in $T_{B\to}$ are added to R, we need to check all non-inert transitions of t to determine whether they are not in $T_{B\to}$ at Lines 2.14ℓ–2.18ℓ, i.e., in the left column of Algorithm 2. Note that checking all successors of such a state is balanced with marking the states in R. In the next section, we explain how to initialize the array *untested* in constant time.

The coroutine that finishes first, provided that its number of states does not exceed $\frac{1}{2}|B|$, has completely computed the smaller subblock resulting from the refinement, and the other coroutine can be aborted. As soon as the number of states of a coroutine is known to exceed $\frac{1}{2}|B|$, it is aborted, and the other coroutine can continue to identify the smaller subblock.

In detail, the runtime complexity of $\langle R, U\rangle := \mathsf{split}(B, T)$ is:

- $O(|R_\to| + |R_\leftarrow|)$, if $|R| \le |U|$, and
- $O(|\mathit{Marked}(T)| + |U_\to| + |U_\leftarrow| + |(\mathit{Bottom}(R) \setminus \mathit{Bottom}(B))_\to|)$, if $|U| \le |R|$.

This complexity can be inferred as follows. First, note that we execute the coroutines in lockstep. That means that running them will incur an overhead that is at most proportional to the faster of the two. As soon as one subblock becomes too large, its coroutine is aborted to reduce the overhead. Therefore, it is enough to show that the runtime bound for the smaller subblock is satisfied. Note that if both subblocks have the same size, $|R| = |U| = \frac{1}{2}|B|$, the faster of the two finishes first, so both runtime bounds apply.

Algorithm 2 Refinement of a block under a splitter

```
2.1: function split(block B, block-bunch-slice T)
2.2: R := B -Marked(T)->                                                          } O(|Marked(T)|)
2.3: U := Bottom(B) \ R                                                           }
2.4: begin coroutines
2.5:     Set untested[t] to undefined for all t∈B  ||  R := R ∪ B -T\Marked(T)->  } O(1) or O(|R→|)
2.6:     for s ∈ U while |U| ≤ ½|B| do  ||  for all s ∈ R while |R| ≤ ½|B| do
2.7:         for all inert transitions t -τ-> s do  ||      for all inert transitions t -τ-> s do
2.8:             if t ∈ R then Skip state t
2.9:             if untested[t] is undefined then                                 } O(|U←|) or
2.10:                untested[t] := |{t -τ-> u | u ∈ B}|                          } O(|R←|)
2.11:            end if
2.12:            untested[t] := untested[t] − 1
2.13:            if untested[t] > 0 then Skip state t
2.14:            if B-T-> ⊈ R then
2.15:                for all non-inert t -α-> u do                                } O(|U→| +
2.16:                    if t -α-> u ∈ T then Skip t                              }   |(Bottom(R) \
2.17:                end for                                                      }     Bottom(B))→|)
2.18:            end if
2.19:            Add t to U  ||  Add t to R                                       } O(|U←|) or
2.20:        end for  ||  end for                                                 } O(|R←|)
2.21:    end for  ||  end for
2.22:    if |U| > ½|B| then  ||  if |R| > ½|B| then
2.23:        Abort this coroutine  ||  Abort this coroutine
2.24:    end if  ||  end if                                                       } O(1)
2.25:    Abort the other coroutine  ||  Abort the other coroutine
2.26:    return (B \ U, U)  ||  return (R, B \ R)                                 }
2.27: end coroutines
```

Consider the case $|R| \leq |U|$. Observe $|Marked(T)| \leq |R_{\rightarrow}|$, and all work is attributed to the coroutine at the right of Algorithm 2, so we get $O(|R_{\rightarrow}| + |R_{\leftarrow}|)$ directly from the R-coroutine.

Now consider the case $|U| \leq |R|$. Then we use time in $O(|Marked(T)|)$ for Lines 2.2–2.3, and we use time in $O(|U_{\leftarrow}|)$ for everything else except Lines 2.14ℓ–2.18ℓ. For these latter lines, we distinguish two cases. If it turns out that t has no transition $t \xrightarrow{\alpha} u \in T$, it is a U-state, so we can attribute the time to $O(|U_{\rightarrow}|)$. Otherwise, i.e., if there is some $t \xrightarrow{\alpha} u \in T$, it is an R-state. How to account for such an R-state in the coroutine that is supposed to find U? The solution is that t is a new bottom state. It had some inert transitions in B, but they all are now in $R \xrightarrow{\tau} U$. So we attribute the time to the outgoing transitions of new bottom states: $O(|(Bottom(R) \setminus Bottom(B))_{\rightarrow}|)$. Note that eventually we reach the situation that all states in $B_{\xrightarrow{T}}$ are in R by Line 2.5r, and this expensive check is skipped.

It can also happen that U is empty. In that case time in $O(|Marked(T)|)$ is spent in split. The initialisation takes $O(|Marked(T)|)$ time, and since U is empty after initialisation, the left coroutine terminates immediately, and the red coroutine is aborted. The latter takes $O(1)$ time.

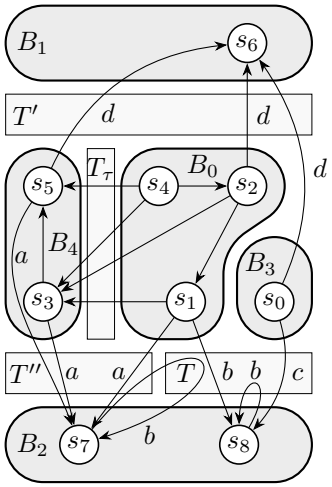
5 Implementation details

This section shows how the operations of the abstract algorithm can be implemented in a way that fits all time bounds.

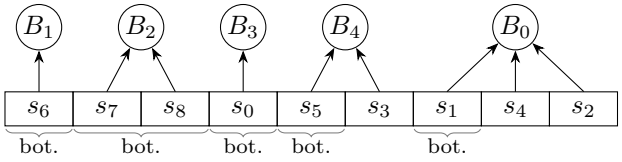
5.1 Implementation of data types

States are stored as a *refinable partition* data structure [20], grouped per block, i.e., in an array such that states in the same block are adjacent to each other. Then, a block of states can be described as a slice in this array. An example of this is shown in Figure 3. Within each such slice,

we separate bottom from non-bottom states and states known to be destined for R from other states. Note that initially, R contains exactly the states with marked outgoing transitions. This is illustrated in the first two lines of Figure 4. This makes it possible to visit all states in a block B in time $O(|B|)$, to find its bottom states that are not in R in constant time (cf. Line 2.3 of Algorithm 2) and visit these bottom states in time $O(|Bottom(B) \setminus R|)$.



(a) An example LTS and partition. Unlabelled transitions are assumed to be τ-transitions.



(b) Refinable partition data structure.

Figure 3: Snapshot of an LTS with its partitions, and the corresponding refinable partition data structure.

Example 5.1. Figure 3a shows an LTS and its current partition. The corresponding refinable partition data structure is shown in Figure 3b. The history of the splitting is as follows. We start with a block consisting of all states. From this, the states that cannot inertly reach a visible transition are split off as B_1. Subsequently, B_2, B_3 and B_4 are split off from the remaining states, resulting in the partition shown in the figure. Note that in Figure 3b it is also illustrated that in blocks B_0 and B_4 the non-bottom states are grouped together, and appear after the bottom states.

When a block is split, we need to update the data structure. This can be done in time proportional to the smaller subblock at Lines 1.17 and 1.25. Figure 4 illustrates how U and R are located in the slice of B. In the third line in the figure, each state with $untested[t] \neq 0$ is stored in a specific slice of non-bottom states. To initialise $untested[t]$ to "undefined" at Line 2.5ℓ of Algorithm 2, it suffices to set this slice to the empty slice. After split has finished, the bottom states of R and the non-bottom states of U exchange places; note that this can be done in time proportional to the smaller of the two subblocks. In the example, a part of the non-bottom states of U can even stay where they are. At the end, new bottom states are searched and added to the bottom states of R.

Transitions are stored in four linked *refinable partitions* [20]. While not all four are essential for the concepts of the algorithm, we need them to ensure the time complexity bounds. In Figure 5 we illustrate each of the refinable partitions for the transitions corresponding to the example in Figure 3.

- Transitions are stored in an array *grouped per bunch,* i.e., transitions in the same bunch are adjacent to each other, see Figure 5a. Then, each bunch can be described as a slice in the array.

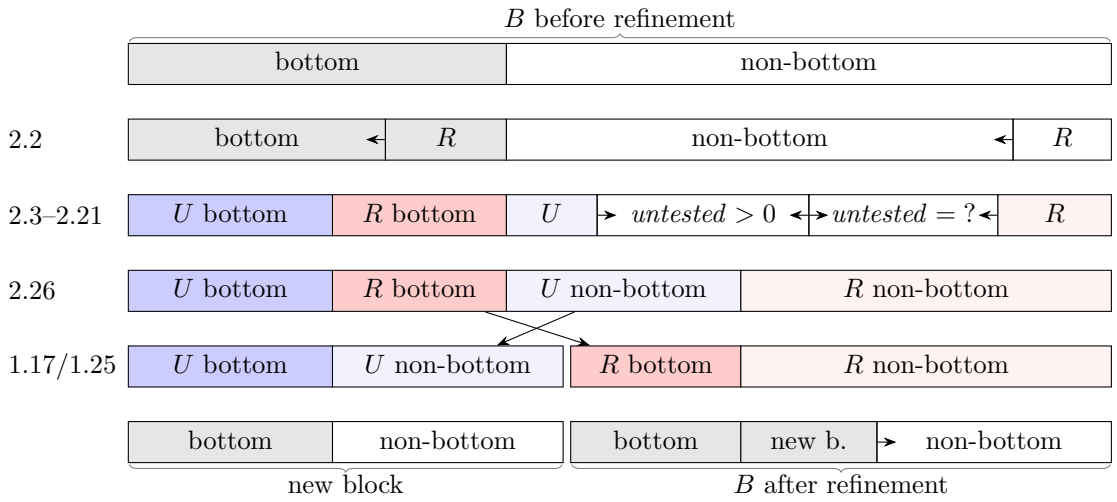


Figure 4: Internal structure of a block during and shortly after split. In this example, the U-subblock is smaller, so it will become the new block.

Within a bunch, transitions are grouped further per action-block-slice. As a consequence, when a small action-block-slice needs to be split off a bunch, one can easily select either the first or the last action-block-slice in the bunch and split it off in constant time.

When a block is split, we need to split its action-block-slices, which can be done in time proportional to the incoming transitions of the smaller subblock at Lines 1.17 and 1.25. This operation fits into the time budget.

- Transitions are stored in an array *grouped per block-bunch-slice,* see Figure 5b. Within each slice, the marked transitions are separated from the unmarked ones when the block-bunch-slice is a splitter.

  When a bunch is split, we need to split its block-bunch-slices, which can be done in time proportional to the smaller new bunch at Line 1.8. When a block is split, we need to split its block-bunch-slices, which can be done in time proportional to the outgoing transitions of the smaller subblock at Lines 1.17 and 1.25. Both operations fit into the allowed time budget.

  We need this partition to mark transitions quickly, namely in constant time per transition, to visit all marked transitions at Line 2.2, and to visit all other transitions at Line 2.5r.

- Transitions are stored *grouped per source state,* see Figure 5c. Within each slice, transitions are further grouped into non-inert and inert transitions, and the non-inert transitions are grouped per bunch.

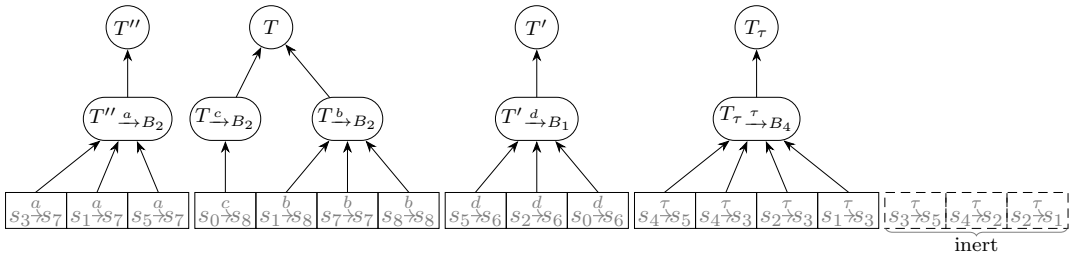
  When a bunch is split, we need to regroup the transitions in that bunch as well. This is done in time proportional to the smaller new bunch at Line 1.8.

  We need this partition to visit all outgoing transitions of a state at Line 2.14ℓ. We also use this partition to decide whether a state with a transition in $T_{\xrightarrow{a}B'}$ also has a transition in $T_{B\to} \setminus T_{\xrightarrow{a}B'}$: While regrouping the transitions in $T_{B\to}$ leaving from the same source state, we can recognize whether all transitions move to $T_{\xrightarrow{a}B'}$ or some remain in $T_{B\to} \setminus T_{\xrightarrow{a}B'}$.

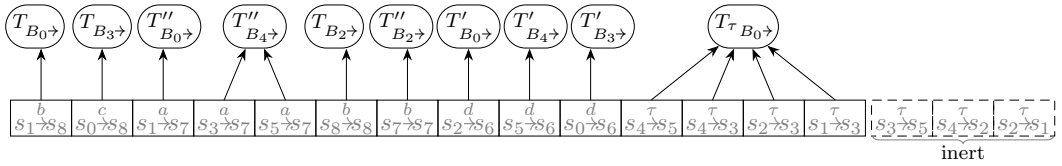
- Transitions are stored *grouped per target state,* see Figure 5d. Within each slice, transitions are further grouped into non-inert and inert transitions. This partition hardly ever changes. We need this partition to visit all incoming (inert) transitions of a state at Line 2.7.

When a transition becomes non-inert, we have to change all four partitions: Create a new bunch, create a new block-bunch-slice, and move the transition from the inert to the non-inert ones in the last two partitions. We do this by running over all outgoing (formerly) inert transitions of R or all incoming (formerly) inert transitions of U, depending on which subblock is smaller. This requires either time $O(|R_{\rightarrow}|)$ or $O(|U_{\leftarrow}|)$, respectively, which fits into the time budget at Lines 1.17 and 1.25.

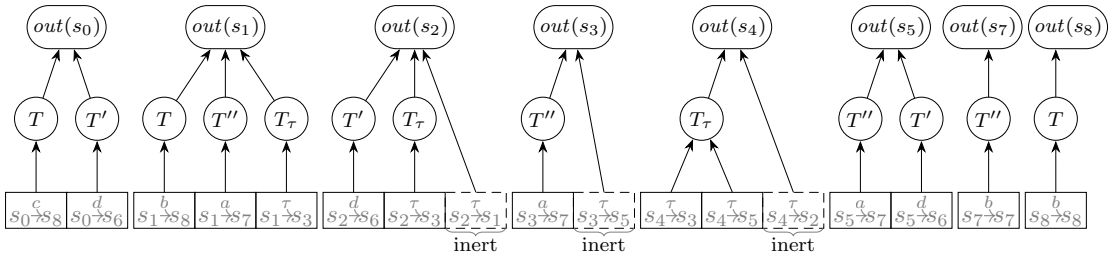
In our implementation, the four transition partitions are linked together via pointers. When source and target state and (pointers to) the relevant slices mentioned above are only stored once, we need nine pointers or `size_t` integers per transition.



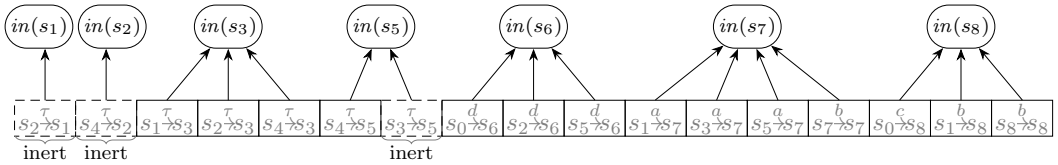
(a) The forest of transitions per bunch, grouped by action-block-slice, with the bunches.



(b) The forest of transitions per block-bunch-slice.



(c) The forest of transitions per source state, grouped into non-inert and inert transitions, and non-inert transitions grouped per bunch.



(d) The forest of transitions per goal state, grouped into non-inert and inert transitions.

Figure 5: Four refinable partition instances for the transitions of the example in Figure 3.

Block-bunch-slices are also stored in lists. We store, per block, a list of its stable block-bunch-slices, and additionally one global list containing all unstable block-bunch-slices, called the splitter list.

When a block is split, its list of stable block-bunch-slices needs to be distributed over the two blocks. This does not require additional time complexity over splitting the block-bunch-slices

themselves. New stable block-bunch-slices are inserted into the list of the new block. New unstable block-bunch-slices are inserted into the splitter list.

We obviously need the unstable block-bunch-slices at Line 1.14, and we need the stable block-bunch-slices of block N at Line 1.27. We store a stability flag with each block-bunch-slice to decide whether a split-off new block-bunch-slice should go into the stable or the unstable list. When executing Line 1.27, we now need to clear the stability flag of every stable block-bunch-slice of N. As every block-bunch-slice of N either already contains a transition from a new bottom state, or will contain a transition from a new bottom state after it has been used as a splitter, we assign the runtime needed to clear this flag to the present and future new bottom states of N.

Care needs to be taken that $T_{U\to} \setminus T_{U\xrightarrow{a}B'}$ can be found at Line 1.19. We ensure this as follows. At Line 1.10, the primary splitter $T_{B\xrightarrow{a}B'}$ and the secondary splitter $T_{B\to} \setminus T_{B\xrightarrow{a}B'}$ are added to the splitter list in this order. After $T_{B\xrightarrow{a}B'}$ has been removed from the list (Line 1.16), $T_{B\to} \setminus T_{B\xrightarrow{a}B'}$ is the first element of the remaining list. At Line 1.17, this is split into $T_{R\to} \setminus T_{R\xrightarrow{a}B'}$ and $T_{U\to} \setminus T_{U\xrightarrow{a}B'}$, in an order that depends on which subblock is the smaller. So either the first or the second element of the splitter list is the required slice at Line 1.19.

5.2 Several small optimisations

We mention a few additional optimisations that our implementation uses, which are not essential for the complexity, but speed up the implementation.

In cases when we mark all transitions in a block-bunch-slice (Lines 1.11 and 1.22), we instead add their source states to R immediately.

In Line 1.27, we actually know that N is stable under $R \xrightarrow{\tau} U$ because that was the splitter applied last, so we do not make $R \xrightarrow{\tau} U$ unstable. Also, if $\mathit{Bottom}(N) \setminus \mathit{Bottom}(R) = \emptyset$, N is stable under $T'_{N\to} \subseteq T'_{B\to}$ (because R was stable under $T'_{B\to}$, and stability is preserved if no more new bottom states are found at Line 1.23), so we do not make $T'_{N\to}$ unstable.

6 Benchmarks

The new algorithm (JGKW19) has been implemented in the mCRL2 toolset [5], and is available in its 201908.0 release. This toolset also contains implementations of various other algorithms, such as the algorithm by Groote and Vaandrager (GV) [10] and the GJKW algorithm of [9], which we refer to as GJKW17. In addition, it offers an implementation of the partition-refinement algorithm using state signatures by Blom and Orzan (BO) [3]. For each state, a signature is maintained describing which blocks the state can reach directly via its outgoing transitions. Although its time complexity is $O(mn^2)$, in some cases, it is known to outperform GV.

In this section, we report on the experiments we have conducted to compare GV, BO, GJKW17 and JGKW19 when applied to practical examples. All experiments involve the branching bisimulation minimisation of a given LTS, which GJKW17 first transforms into a Kripke structure. Note that for an LTS of n states and m transitions, this transformation results in a Kripke structure consisting of $n + m$ states and $2m$ transitions in the worst case.

The set of benchmarks consists of all LTSs offered by the VLTS benchmark set[2] with at least 60,000 transitions, plus three cases that have been derived from models distributed with the mCRL2 toolset. These models are:

1. **lift6-final**: this model is based on an elevator model, extended to six elevators;
2. **dining_14**: this is the dining philosophers model with 14 philosophers;
3. **1394-fin3**: this model is an altered version of the 1394-fin model, extended to three processes and two data elements.

[2] `http://cadp.inria.fr/resources/vlts`.

Table 1: Structural characteristics of the benchmark LTSs.

model	n	m	m_τ	$\lvert Act\rvert$	min. n	min. m
vasy_40_60	40,006	60,007	20,003	4	20,003	40,004
vasy_18_73	18,746	73,043	39,217	18	2,326	9,751
vasy_157_297	157,604	297,000	31,798	236	3,038	12,095
vasy_52_318	52,268	318,126	130,752	18	66	333
vasy_83_325	83,436	325,584	45,696	212	42,195	197,200
vasy_116_368	116,456	368,569	263,296	22	22,398	87,674
vasy_720_390	720,247	390,999	1	50	3,278	116,537
vasy_69_520	69,754	520,633	1	136	69,753	520,632
cwi_371_641	371,804	641,565	445,600	62	2,134	5,634
vasy_166_651	166,464	651,168	91,392	212	42,195	197,200
cwi_214_684	214,202	684,419	550,611	6	478	1,612
cwi_142_925	142,472	925,429	862,298	8	23	49
vasy_386_1171	386,496	1,171,872	122,976	74	71	108
vasy_66_1302	66,929	1,302,664	117,866	82	51,128	1,018,692
vasy_164_1619	164,865	1,619,204	109,910	38	992	3,456
vasy_65_2621	65,537	2,621,480	0	72	65,536	2,621,440
cwi_566_3984	566,640	3,984,157	3,666,614	12	198	791
vasy_1112_5290	1,112,490	5,290,860	0	23	265	1,300
cwi_2165_8723	2,165,446	8,723,465	3,830,225	27	4,256	20,880
vasy_6120_11031	6,120,718	11,031,292	3,152,976	126	2,505	5,358
vasy_2581_11442	2,581,374	11,442,382	2,508,518	224	704,737	3,972,600
vasy_574_13561	574,057	13,561,040	0	141	3,577	16,168
vasy_4220_13944	4,220,790	13,944,372	2,546,649	224	1,186,266	6,863,329
vasy_4338_15666	4,338,672	15,666,588	3,127,116	224	704,737	3,972,600
cwi_2416_17605	2,416,632	17,605,592	17,490,904	16	730	2,899
vasy_6020_19353	6,020,550	19,353,474	17,526,144	512	256	510
vasy_11026_24660	11,026,932	24,660,513	2,748,559	120	775,618	2,454,834
lift6-final	6,047,527	26,539,368	12,668,580	31	1,699	9,870
vasy_12323_27667	12,323,703	27,667,803	3,153,502	120	876,944	2,780,022
vasy_8082_42933	8,082,905	42,933,110	2,535,944	212	290	680
cwi_7838_59101	7,838,608	59,101,007	22,842,122	21	62,031	470,230
dining_14	18,378,370	164,329,284	142,722,790	15	228,486	2,067,856
cwi_33949_165318	33,949,609	165,318,222	74,133,306	32	12,463	71,466
1394-fin3	126,713,623	276,426,688	172,900,987	104	160,258	538,936

Table 1 presents the structural characteristics for each benchmark: the number of states (n), the number of transitions (m), the number of τ-transitions (m_τ), the number of actions ($|Act|$), and the number of states and transitions after branching bisimulation reduction (min. n and min. m, respectively).

All experiments have been conducted on individual nodes of the DAS-5 cluster [1]. Each of these nodes was running CentOS Linux 7.4, had an Intel Xeon E5-2698-v3 2.3GHz CPU, and was equipped with 256 GB RAM. The experiments were performed using development version 201808.0.c59cfd413f of mCRL2.[3]

Table 2 presents the obtained results. On each benchmark, we have applied each of the four algorithms ten times, and report the mean runtime (in seconds or minutes) and memory use (in MB or GB) of those ten runs. In the table, only the significant digits are listed, which are identified by first estimating the standard deviation, given the ten results. Given results $x_0, \ldots, x_9$, the standard deviation *std* is estimated to be $std = \frac{(\Sigma_{0\leq i\leq 9} x_i^2) - (\Sigma_{0\leq i\leq 9} x_i)^2/10}{8.5}$ [4]. For all presented data the estimated standard deviation is less than 20% of the mean. Cases in which this is not true are indicated by '-' in Table 2.

Concerning the significant digits, a decimal dot indicates that the unit digit is significant. If this dot is missing, there is one insignificant zero. The estimated standard deviation is used to identify the significant digits. For example, '3.6 s' has a standard deviation in $[0.01, 0.1)$ s, '540. MB' has a standard deviation in $[0.1, 1)$ MB, and '100 s' has a standard deviation in $[1, 10)$ s.

The ▼-symbol after a table entry indicates that the measurement is significantly better than the corresponding measurements for the other three algorithms, and the ▲-symbol indicates that the measurement is significantly worse. Here, the results are considered significant if, given a hundred tables such as Table 2, one table of running time (resp. memory) is expected to contain spuriously significant results.

[3] `https://github.com/mCRL2org/mCRL2/commit/c59cfd413f`

Table 2: Running time and memory use results for GV, BO, GJKW17 and JGKW19. ▼and ▲: significantly better (resp. worse) than all three other algorithms.

model	time				space			
	GV	BO	GJKW17	JGKW19	GV	BO	GJKW17	JGKW19
vasy_40_60	24. s	138. s ▲	.1 s	.05 s ▼	65.5 MB	60.6 MB	70 MB	60 MB
vasy_18_73	.21 s	.37 s ▲	.11 s	.07 s ▼	55.6 MB	56.7 MB	50 MB	50 MB
vasy_157_297	1.7 s	2. s	.4 s	.2 s ▼	97.3 MB	94.3 MB	127.2 MB ▲	90 MB
vasy_52_318	.31 s	.9 s ▲	.2 s	.2 s	73.4 MB	90.4 MB	90.6 MB ▲	73.4 MB
vasy_83_325	2.6 s ▲	1.0 s	.9 s	.3 s ▼	116.2 MB	.11 GB	230.5 MB ▲	.10 GB
vasy_116_368	.9 s	5. s ▲	.6 s	.4 s ▼	92.8 MB	110.6 MB	.13 GB ▲	90 MB
vasy_720_390	.4 s	.9 s	.6 s	.4 s	105.2 MB	103.2 MB	.19 GB ▲	95.9 MB ▼
vasy_69_520	1.5 s	4. s ▲	2.4 s	.8 s ▼	.15 GB	.15 GB	358.1 MB ▲	162.0 MB
cwi_371_641	7.4 s ▲	5.9 s	1. s	.7 s	.17 GB	229.0 MB ▲	185.4 MB	.14 GB ▼
vasy_166_651	4.9 s ▲	1.9 s	2. s	.7 s ▼	157.5 MB	141.8 MB	342.9 MB ▲	139.5 MB ▼
cwi_214_684	1.4 s	9. s ▲	.5 s	.5 s	140.7 MB	162.1 MB ▲	152.0 MB	.13 GB
cwi_142_925	1.4 s ▲	.8 s	1.0 s	.9 s	152.5 MB	117.9 MB ▼	156.6 MB ▲	152.5 MB
vasy_386_1171	1.4 s	2. s ▲	1.3 s	.9 s ▼	229.2 MB	210.1 MB ▼	273.4 MB ▲	228.7 MB
vasy_66_1302	3. s	4.7 s	5. s	2.2 s ▼	.23 GB ▼	283.1 MB	618.1 MB ▲	268.0 MB
vasy_164_1619	2.0 s	5. s ▲	3. s	-	.25 GB	235.4 MB	262.4 MB	245.0 MB
vasy_65_2621	90 s ▲	11. s	20 s	4.7 s ▼	.5 GB	534.7 MB	1.8 GB ▲	.5 GB
cwi_566_3984	8. s	7. s	8. s	6. s	.5 GB	351.5 MB ▼	514.0 MB ▲	.5 GB
vasy_1112_5290	10. s	17. s ▲	10 s	10 s	.8 GB	720.9 MB	931.5 MB	.7 GB
cwi_2165_8723	.4 min	3. min ▲	-	.3 min	1.3 GB	1.8726 GB	2.1321 GB ▲	1.2 GB
vasy_6120_11031	2. min ▲	1.7 min	-	.4 min	1.8 GB	1.7379 GB	3.6596 GB ▲	1.5960 GB ▼
vasy_2581_11442	10 min ▲	3. min	-	-	1.5999 GB	1.7434 GB	4.1299 GB ▲	1.4 GB ▼
vasy_574_13561	50 s	56. s	-	-	1.8835 GB ▲	1.5217 GB	1.5 GB	1.5 GB
vasy_4220_13944	30 min ▲	5. min	-	.6 min	2.0965 GB	2.3188 GB	5.8661 GB ▲	2.0 GB ▼
vasy_4338_15666	34. min ▲	3. min	2. min	.8 min ▼	2.4043 GB	2.3559 GB	5.9888 GB ▲	1.8535 GB ▼
cwi_2416_17605	30 s	19. s	20 s	20 s	1.6 GB	1.5157 GB ▼	1.6638 GB	1.6748 GB ▲
vasy_6020_19353	25. s	40 s ▲	6. s	5. s	870. MB	2.3442 GB ▲	870. MB	870. MB
vasy_11026_24660	50 min ▲	20 min	3. min	1. min ▼	3.6475 GB	4.0513 GB	9.6425 GB ▲	3.4412 GB ▼
lift6-final	1.0 min	3. min ▲	-	.9 min	3.3846 GB	8.1984 GB ▲	6.2971 GB	3.2125 GB ▼
vasy_12323_27667	40 min ▲	10 min	-	1. min	4.0091 GB	4.5371 GB	10.6743 GB ▲	3.7298 GB ▼
vasy_8082_42933	2. min	5. min ▲	-	2. min	6.1231 GB	5.4358 GB ▼	6.6896 GB ▲	5.4600 GB
cwi_7838_59101	5. min	100 min ▲	6. min	3. min	6.5283 GB ▼	8.3266 GB	13.7899 GB ▲	6.7646 GB
dining_14	17. min	20 min	20 min	10 min	20.4826 GB ▼	21.7156 GB	23.7810 GB ▲	20.9756 GB
cwi_33949_165318	11. min	80 min ▲	20 min	8. min	22.7204 GB	33.0351 GB	37.8606 GB ▲	21.0611 GB ▼
1394-fin3	25. h ▲	3. h	.5 h	.3 h ▼	37.4893 GB	71.8698 GB ▲	53.2166 GB	31.5132 GB ▼
Total	**28. h ▲**	**8. h**	**1.4 h**	**.8 h ▼**	**121.8 GB**	**176.33 GB**	**194.2 GB ▲**	**112.0 GB ▼**

Concerning the runtimes, clearly, GV and BO perform significantly worse than the other two algorithms, and JGKW19 in many cases performs significantly better than the others. In particular, it should be noted that, although GJKW17 has the same time complexity, JGKW19 often still outperforms GJKW17. Concerning memory use, in the majority of cases GJKW17 uses more memory than the others, while sometimes BO is the most memory-hungry. JGKW19 is much more competitive, in many cases even outperforming every other algorithm.

Overall, the results demonstrate that when applied to practical cases, JGKW19 is generally the most efficient algorithm time-wise, and when other algorithms have similar runtimes, it is almost always the most efficient memory-wise. This combination makes JGKW19, the algorithm presented in this paper, currently the best option for branching bisimulation minimisation of LTSs.

References

[1] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *IEEE Computer*, 49(5):54–63, 2016.

[2] Maarten Bartholomeus, Bas Luttik, and Tim Willemse. Modelling and analysing ERTMS Hybrid Level 3 with the mCRL2 toolset. In *Formal Methods for Industrial Critical Systems*, pages 98–114. Springer, Cham, September 2018.

[3] S.C.C. Blom and S. Orzan. Distributed branching bisimulation reduction of state spaces. *Electron. Notes Theor. Comput. Sci.*, 80(1):99–113, 2003.

[4] R.M. Brugger. A note on unbiased estimation of the standard deviation. *The American Statistician*, 23(4):32, 1969.

[5] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability. In Tomás Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II*, volume 11428 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2019.

[6] Rocco De Nicola and Frits Vaandrager. Three Logics for Branching Bisimulation. *J. ACM*, 42(2):458–487, March 1995.

[7] Rob. J. van Glabbeek. The linear time — branching time spectrum II. In *CONCUR'93*, pages 66–81. Springer, Berlin, Heidelberg, August 1993.

[8] Rob J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *J. ACM*, 43(3):555–600, 1996.

[9] Jan Friso Groote, David N. Jansen, Jeroen J. A. Keiren, and Anton J. Wijs. An $O(m \log n)$ algorithm for computing stuttering equivalence and branching bisimulation. *ACM Trans. Comput. Logic*, 18(2):Article 13, 2017.

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

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

[12] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 453–462, October 1995.

[13] John Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Zvi Kohavi and Azaria Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, January 1971.

[14] Paris C. Kanellakis and Scott A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 228–240, New York, 1983. ACM.

[15] Paris C. Kanellakis and Scott A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, 1990.

[16] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[17] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.

[18] Michel A. Reniers, Rob Schoren, and Tim A. C. Willemse. Results on Embeddings Between State-Based and Event-Based Systems. *The Computer Journal*, 57(1):73–92, January 2014.

[19] Antti Valmari. Bisimilarity minimization in $O(m \log n)$ time. In Giuliana Franceschinis and Karsten Wolf, editors, *Applications and theory of Petri nets: PETRI NETS*, volume 5606 of *LNCS*, pages 123–142. Springer, Berlin, 2009.

[20] Antti Valmari and Petri Lehtinen. Efficient minimization of DFAs with partial transition functions. In Susanne Albers and Pascal Weil, editors, *25th international symposium on theoretical aspects of computer science: STACS*, volume 1 of *LIPIcs*, pages 645–656. Schloss Dagstuhl, Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2008.

A Branching bisimilarity via translation to Kripke structures is $O(m \log n)$

In the original paper [9], it is stated that the complexity of determining branching bisimilarity is $O(m(\log |Act| + \log n))$ when using the following translation from LTS to Kripke structure, and subsequently computing divergence-blind stuttering bisimilarity.

Definition A.1 (LTS Embedding [6])**.** 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. $S_A = S \cup \{\langle a, t\rangle \mid \exists s \in S. s \xrightarrow{a} t\}$
2. $AP = Act \cup \{\bot\}$
3. $\rightarrow$ is the least relation satisfying (for $s, t \in S$, $a \in Act \setminus \{\tau\}$)

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

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

The observations made were as follows: an LTS with n states and m transitions, in the worst case, has an embedding of $n + m$ states and $2m$ transitions, so the algorithm requires $O(m \log(n + m))$, or, since $m \leq |Act|\, n^2$, $O\big(m \log(n + |Act|\, n^2)\big) = O(m(\log |Act| + \log n))$ time.

However, if we consider the analysis of the complexity of the algorithm for divergence-blind stuttering bisimilarity a bit more closely, we can observe that the actual complexity is $O(m \log b)$, where b is the number of states in the largest block in the initial partition.

For the Kripke structure that is the result of translating the LTS, let $S_a = \{s \in S_A \mid L(s) = \{a\}\}$ and observe that $|S_a| \leq |S|$ for all $a \in Act$. The initial partition in the divergence-blind stuttering bisimilarity algorithm is such that s and t are in the same block iff $L(s) = L(t)$. Therefore, the largest block in the initial partition of a translated Kripke structure has at most $|S| = n$ states. Hence, $b = n$. The complexity of deciding branching bisimilarity via reduction to Kripke structure and using divergence-blind stuttering bisimilarity is thus $O(m \log n)$.

Science Reports **Department of Mathematics and Computer Science**
Technische Universiteit Eindhoven

If you want to receive reports, send an email to: wsinsan@tue.nl (we cannot guarantee the availability of the requested reports).

In this series appeared (from 2012):

12/01	S. Cranen	Model checking the FlexRay startup phase
12/02	U. Khadim and P.J.L. Cuijpers	Appendix C / G of the paper: Repairing Time-Determinism in the Process Algebra for Hybrid Systems ACP
12/03	M.M.H.P. van den Heuvel, P.J.L. Cuijpers, J.J. Lukkien and N.W. Fisher	Revised budget allocations for fixed-priority-scheduled periodic resources
12/04	Ammar Osaiweran, Tom Fransen, Jan Friso Groote and Bart van Rijnsoever	Experience Report on Designing and Developing Control Components using Formal Methods
12/05	Sjoerd Cranen, Jeroen J.A. Keiren and Tim A.C. Willemse	A cure for stuttering parity games
12/06	A.P. van der Meer	CIF MSOS type system
12/07	Dirk Fahland and Robert Prüfer	Data and Abstraction for Scenario-Based Modeling with Petri Nets
12/08	Luc Engelen and Anton Wijs	Checking Property Preservation of Refining Transformations for Model-Driven Development
12/09	M.M.H.P. van den Heuvel, M. Behnam, R.J. Bril, J.J. Lukkien and T. Nolte	Opaque analysis for resource-sharing components in hierarchical real-time systems - extended version –
12/10	Milosh Stolikj, Pieter J. L. Cuijpers and Johan J. Lukkien	Efficient reprogramming of sensor networks using incremental updates and data compression
12/11	John Businge, Alexander Serebrenik and Mark van den Brand	Survival of Eclipse Third-party Plug-ins
12/12	Jeroen J.A. Keiren and Martijn D. Klabbers	Modelling and verifying IEEE Std 11073-20601 session setup using mCRL2
12/13	Ammar Osaiweran, Jan Friso Groote, Mathijs Schuts, Jozef Hooman and Bart van Rijnsoever	Evaluating the Effect of Formal Techniques in Industry
12/14	Ammar Osaiweran, Mathijs Schuts, and Jozef Hooman	Incorporating Formal Techniques into Industrial Practice
13/01	S. Cranen, M.W. Gazda, J.W. Wesselink and T.A.C. Willemse	Abstraction in Parameterised Boolean Equation Systems
13/02	Neda Noroozi, Mohammad Reza Mousavi and Tim A.C. Willemse	Decomposability in Formal Conformance Testing
13/03	D. Bera, K.M. van Hee and N. Sidorova	Discrete Timed Petri nets
13/04	A. Kota Gopalakrishna, T. Ozcelebi, A. Liotta and J.J. Lukkien	Relevance as a Metric for Evaluating Machine Learning Algorithms
13/05	T. Ozcelebi, A. Weffers-Albu and J.J. Lukkien	Proceedings of the 2012 Workshop on Ambient Intelligence Infrastructures (WAmIi)
13/06	Lotfi ben Othmane, Pelin Angin, Harold Weffers and Bharat Bhargava	Extending the Agile Development Process to Develop Acceptably Secure Software
13/07	R.H. Mak	Resource-aware Life Cycle Models for Service-oriented Applications managed by a Component Framework
13/08	Mark van den Brand and Jan Friso Groote	Software Engineering: Redundancy is Key
13/09	P.J.L. Cuijpers	Prefix Orders as a General Model of Dynamics

14/01	Jan Friso Groote, Remco van der Hofstad and Matthias Raffelsieper	On the Random Structure of Behavioural Transition Systems
14/02	Maurice H. ter Beek and Erik P. de Vink	Using mCRL2 for the analysis of software product lines
14/03	Frank Peeters, Ion Barosan, Tao Yue and Alexander Serebrenik	A Modeling Environment Supporting the Co-evolution of User Requirements and Design
14/04	Jan Friso Groote and Hans Zantema	A probabilistic analysis of the Game of the Goose
14/05	Hrishikesh Salunkhe, Orlando Moreira and Kees van Berkel	Buffer Allocation for Real-Time Streaming on a Multi-Processor without Back-Pressure
14/06	D. Bera, K.M. van Hee and H. Nijmeijer	Relationship between Simulink and Petri nets
14/07	Reinder J. Bril and Jinkyu Lee	CRTS 2014 - Proceedings of the 7th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems
14/08	Fatih Turkmen, Jerry den Hartog, Silvio Ranise and Nicola Zannone	Analysis of XACML Policies with SMT
14/09	Ana-Maria Şutîi, Tom Verhoeff and M.G.J. van den Brand	Ontologies in domain specific languages – A systematic literature review
14/10	M. Stolikj, T.M.M. Meyfroyt, P.J.L. Cuijpers and J.J. Lukkien	Improving the Performance of Trickle-Based Data Dissemination in Low-Power Networks
15/01	Önder Babur, Tom Verhoeff and Mark van den Brand	Multiphysics and Multiscale Software Frameworks: An Annotated Bibliography
15/02	Various	Proceedings of the First International Workshop on Investigating Dataflow In Embedded computing Architectures (IDEA 2015)
15/03	Hrishikesh Salunkhe, Alok Lele, Orlando Moreira and Kees van Berkel	Buffer Allocation for Realtime Streaming Applications Running on a Multi-processor without Back-pressure
15/04	J.G.M. Mengerink, R.R.H. Schiffelers, A. Serebrenik, M.G.J. van den Brand	Evolution Specification Evaluation in Industrial MDSE Ecosystems
15/05	Sarmen Keshishzadeh and Jan Friso Groote	Exact Real Arithmetic with Pertubation Analysis and Proof of Correctness
15/06	Jan Friso Groote and Anton Wijs	An $O(m \log n)$ Algorithm for Stuttering Equivalence and Branching Bisimulation
17/01	Ammar Osaiweran, Jelena Marincic Jan Friso Groote	Assessing the quality of tabular state machines through metrics
17/02	J.F. Groote and e.P. de Vink	Problem solving using process algebra considered insightful
18/01	L. Sanchez, W. Wesselink and T.A.C. Willemse	BDD-Based Parity Game Solving: A comparison of Zielonka's Recursive Algorithm, Priority Promotion and Fixpoint Iteration
18/02	Mahmoud Talebi	First-order Closure Approximations for Middle-Sized Systems with Non-linear Rates
18/03	Thomas Neele, Tim A.C. Willemse and Jan Friso Groote	Solving Parameterised Boolean Equation Systems with Infinite Data Through Quotienting (Technical Report)
18/04	Daan Leermakers, Behrooz Razeghi, Shideh Rezaeifar, Boris Škorić, Olga Taran Slava Voloshynovskiy	Optical PUF statistics
19/01	Maurice Laveaux, Jan Friso Groote and Tim A.C. Willemse	Correct and Efficient Antichain Algorithms for Refinement Checking
19/02	Thomas Neel, Tim A.C. Willemse and Wieger Wesselink	Partial-Order Reduction for Parity Games with an Application on Parameterised Boolean Equation Systems (Technical Report)
19/03	David N. Jansen, Jan Friso Groote, Jeroen J.A. Keiren and Anton Wijs	A simpler $O(m \log n)$ algorithm for branching bisimilarity on labelled transition systems