A State-Based $O(m \log n)$ Partitioning Algorithm for Branching Bisimilarity

Jan Friso Groote
Department of Computer Science, Eindhoven University of Technology, The Netherlands

David N. Jansen
Key Laboratory of System Software, Chinese Academy of Sciences, Beijing, China
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

Abstract

We present a new $O(m \log n)$ algorithm to calculate branching bisimulation equivalence, which is the finest commonly used behavioural equivalence on labelled transition systems that takes the internal action τ into account. This algorithm combines the simpler data structure of an earlier algorithm for Kripke structures (without action labels) with the memory-efficiency of a later algorithm partitioning sets of labelled transitions. It employs a particularly elegant four-way split of blocks of states, which refines a block under two splitters and isolates all new bottom states, simultaneously. Benchmark results show that this new algorithm outperforms the best known algorithm for branching bisimulation both in time and space.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Software and its engineering → Formal software verification

Keywords and phrases Algorithm, Branching bisimulation, Partition refinement of states

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.18

Funding *David N. Jansen*: This work is supported by ISCAS Basic Research ISCAS-JCZD-202302 and is part of the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant no. 101008233.

1 Introduction

Branching bisimulation relates behaviourally equivalent states in labelled transition systems that represent behaviour [4]. It takes internal or invisible steps τ into account and preserves the branching structure of processes. It is slightly finer than weak bisimulation [11].

Almost immediately after the discovery of branching bisimulation a decently efficient $O(mn)$-algorithm (where m is the number of transitions and n is the number of states) became available [7]. This was a great asset not only because it allowed to decide quickly whether states were branching bisimilar, but also because it could be used as an efficient preprocessing step for many other behaviour equivalences that take internal steps into account.

A major improvement was the development of an $O(m \log n)$ algorithm [8, 5], working on Kripke structures, which can be regarded as labelled transition systems where all transitions are labelled with τ, as, at the time, it was not clear how to efficiently deal with multiple transition labels. This algorithm could be applied to arbitrary labelled transition systems by translating them to Kripke structures where one extra state was introduced for every transition in the original transition system [1]. As the number of transitions is generally much larger than the number of states, this is costly, both in time and space.

Following [14], where it was proposed to use equivalence classes of transitions to avoid redoing work that has already been done, a new algorithm was developed that works directly on labelled transition systems [10]. By avoiding the costly translation to Kripke structures, this algorithm was 40% faster than the best algorithm known until then.


© Jan Friso Groote and David N. Jansen;
licensed under Creative Commons License CC-BY 4.0

However, the use of equivalence classes of transitions feels unpleasant, given that there are in general far more transitions than states. So, the purpose of this paper is to design an algorithm that avoids partitioning transitions. The essential step that enables us to do so is the efficient grouping of transitions with the same transition label. Instead of the complex trickery used in previous algorithms, we employ a very natural and elegant four-way split of blocks, which is an extension of the three-way split used in the classical algorithm for strong bisimulation on Kripke structures by Paige and Tarjan [12, 2]. The fourth block collects the infamous "new bottom states", that invalidate the stability invariant, allowing us to isolate the work to repair this invariant, avoiding much of the complexity of earlier algorithms.

The result is a new algorithm that is not only simpler and more elegant, but also faster and more memory-efficient than all its predecessors, although it has the same $O(m \log n)$ time complexity – here we assume that the number of action labels is smaller than the number of transitions. Especially on large transition systems with many transitions a 40% reduction in memory usage and a fourfold reduction in time appears to be possible.

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 m is always finite.

It is common to write $s \xrightarrow{a} s'$ for $(s, a, s') \in \rightarrow$. We write $s \xrightarrow{a} s' \in T$ instead of $(s, a, s') \in T$ for $T \subseteq \rightarrow$. We also write $s \xrightarrow{a} S'$ for the set of transitions $\{s \xrightarrow{a} s' \mid s' \in S'\}$, and likewise $S_1 \xrightarrow{a} S_2$ for the set $\{s_1 \xrightarrow{a} s_2 \mid s_1 \in S_1 \text{ and } s_2 \in S_2\}$. We refer to all actions except τ as the visible actions. The transitions labelled with τ are called invisible. If $s \xrightarrow{a} s'$, we say that from s, the state s', the action a, and the transition $s \xrightarrow{a} s'$ are *reachable.* We write $in(s)$ for the incoming transitions of state s and $out(s)$ for the outgoing transitions of s. Likewise, we write $in(B)$ and $out(B)$ for all incoming and outgoing transitions of a set $B \subseteq S$.

▶ **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 $s \mathrel{R} t$ and all transitions $s \xrightarrow{a} s'$ we have:

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

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

If we restrict the definition of branching bisimilarity such that states are only related if they both have, or both have not an infinite sequence of τ's through branching bisimilar states, we obtain the notion of *divergence-preserving* branching bisimulation. This notion is useful as it – contrary to branching bisimilarity – preserves liveness properties. The algorithm that is presented below can be used for divergence-preserving branching bisimilarity with only a minor modification.

Given an equivalence relation R, a transition $s \xrightarrow{a} t$ is called *R-inert* iff $a = \tau$ and $s \mathrel{R} t$. If $t \xrightarrow{\tau} t_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} t_{n-1} \xrightarrow{\tau} t_n \xrightarrow{a} t'$ such that $t \mathrel{R} t_i$ 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 *R-inertly reachable* from t. Note that we can use these notions in combination with (divergence-preserving) branching bisimilarity as they are all equivalence relations.

3 The Main Algorithm

In this section we give a description of the main part of the algorithm to determine the branching bisimulation equivalence classes of a given LTS $(S, Act, \rightarrow)$.

Removal of Inert Loops. As the first step of the algorithm the LTS is preprocessed to contract each τ-strongly connected component (SCC) into a single state. This step is valid since all states in a τ-SCC are branching bisimilar. For divergence-preserving branching bisimilarity, the τ-self-loops are replaced by a special non-internal action. In both cases, τ-self-loops can be suppressed and therefore, from this point onwards, as the algorithm does not change the structure of the LTS further, all τ-paths in the LTS are finite, formalised by the following lemma.

▶ **Lemma 3.1** (No τ-loops)**.** After contracting τ-SCCs into single states, there is no τ-loop in the LTS, i.e., for every sequence $s_1 \xrightarrow{\tau} s_2 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_n$ it holds that $s_i \neq s_j$ for all $1 \leq i < j \leq n$.

In every set of states $B \subseteq S$, the states without τ-transition to another state in B are called *bottom states*. The set of bottom states in B is denoted by $Bottom(B)$. Lemma 3.1 implies the following lemma:

▶ **Lemma 3.2.** For every nonempty set of states $B \subseteq S$, we have
1. $Bottom(B) \neq \emptyset$.
2. For every state $s \in B$, there is a path of τ-transitions within B leading to a bottom state.

Partition Refinement. The algorithm is a partition refinement algorithm of sets of states where two partitions of states are iteratively refined.

▶ **Definition 3.3** (Partition)**.** For a set X a partition $\mathcal{P}$ of X is a disjoint cover of X, i.e., $\mathcal{P} = \{B_i \subseteq X \mid B_i \neq \emptyset, 1 \leq i \leq k\}$ 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 $\mathcal{Q}$ is a *refinement* of $\mathcal{P}$, and $\mathcal{P}$ is *coarser* than $\mathcal{Q}$, iff for every $B' \in \mathcal{Q}$ there is some $B \in \mathcal{P}$ such that $B' \subseteq B$.

A partition induces an equivalence relation in the following way: $s \equiv_{\mathcal{P}} t$ iff there is some $B \in \mathcal{P}$ containing both s and t. We call a transition $\mathcal{P}$-inert iff it is $\equiv_{\mathcal{P}}$-inert, i.e., exactly if it is a transition $s \xrightarrow{\tau} t$ with $s, t \in P$ for some $P \in \mathcal{P}$.

Our algorithm uses two partitions $\mathcal{B}$ and $\mathcal{C}$ of states. The main partition $\mathcal{B}$ contains *blocks,* typically denoted with the letter B, recording 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, formulated contrapositionally by the following invariant:

▶ **Invariant 3.4** ($\mathcal{B}$ preserves branching bisimilarity)**.** For all states $s, t \in S$, if $s \leftrightarrows_b t$, then there is some block $B \in \mathcal{B}$ such that $s, t \in B$.

Stability. We desire to make the partition $\mathcal{B}$ "stable" in the sense that blocks of $\mathcal{B}$ cannot be split further as all states in each block are branching bisimilar. This notion of stability is defined as follows. Consider two sets of states $B, B' \subseteq S$. We say that two states $s, t \in B$ are *stable under* B' iff if $s \xrightarrow{a} s'$ for any $s' \in B'$ then
- either $B = B'$ and $a = \tau$,
- or there are $t_1, \ldots, t_k \in B$ and $t' \in B'$ such that $t = t_1$, $k \geq 1$ and $t_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} t_k \xrightarrow{a} t'$.

A partition $\mathcal{B}$ of S is *stable under* a set of states $B' \subseteq S$ iff for all $B \in \mathcal{B}$ and $s, t \in B$ it holds that s and t are stable under B'. If $\mathcal{B}$ is stable under every $B \in \mathcal{B}$ we say that $\mathcal{B}$ is *stable*.

An important property is that if a partition $\mathcal{B}$ is stable, then the induced equivalence relation $\equiv_{\mathcal{B}}$ is a branching bisimulation.

Constellations. In order to remember which instabilities have already been resolved we introduce a second partition $\mathcal{C}$ of the set of states S of which $\mathcal{B}$ is a refinement. We call the blocks in $\mathcal{C}$ *constellations* and we typically denote constellations with the letter C.

The partition $\mathcal{C}$ records the current knowledge about stability by guaranteeing that $\mathcal{B}$ is always stable under each constellation $C \in \mathcal{C}$. Using Lemma 3.2 this follows from the following invariant.

▶ **Invariant 3.5** ($\mathcal{B}$ is stable under the constellations in $\mathcal{C}$)**.** If there is a transition $s \xrightarrow{a} t$ with $t \in C$ for some constellation $C \in \mathcal{C}$ and $s \in B$ for some block $B \in \mathcal{B}$, and the transition is not $\mathcal{C}$-inert (i.e., $a \neq \tau$ or $s \notin C$), then every bottom state $s' \in B$ has a transition in $s' \xrightarrow{a} C$.

The goal of the algorithm is to let the partition of constellations $\mathcal{C}$ and the partition of blocks $\mathcal{B}$ coincide. If $\mathcal{B} = \mathcal{C}$, then $\mathcal{B}$ is stable, and, using Invariants 3.4 and 3.5, the partition $\mathcal{B}$ exactly characterises branching bisimulation. As this is the core purpose of our algorithm, we formulate it as a theorem, although the proof is straightforward.

▶ **Theorem 3.6.** If $\mathcal{B} = \mathcal{C}$, then $\equiv_{\mathcal{B}} = \leftrightarrows_b$.

The main idea of the algorithm is to refine $\mathcal{C}$ until it coincides with $\mathcal{B}$. After refining $\mathcal{C}$, one may have to refine $\mathcal{B}$ to reestablish Invariant 3.5. For this to work, we use two essential subroutines, namely four-way-split$\mathcal{B}$ (Algorithm 2) and stabilise$\mathcal{B}$ (Algorithm 3). We first give a summary of what these functions do, after which we explain the overall algorithm. In Sections 4 and 5 their underlying algorithms are explained.

The function four-way-split$\mathcal{B}(B, \mathit{SmallSp}, \mathit{LargeSp})$ splits a block B in $\mathcal{B}$ in at most four sub-blocks based on two disjoint sets of transitions *SmallSp* and *LargeSp* that start in B. We require that at least one of these two sets is non-empty. If both are non-empty, every state in B can $\mathcal{B}$-inertly reach a transition in at least one of the two. The four sub-blocks are the following:

AvoidLrg: If $\mathit{LargeSp} \neq \varnothing$, AvoidLrg contains the states in B that cannot $\mathcal{B}$-inertly reach *LargeSp*. If $\mathit{LargeSp} = \varnothing$, then AvoidLrg $= \varnothing$.

AvoidSml: If $\mathit{SmallSp} \neq \varnothing$, AvoidSml contains the states in B that cannot $\mathcal{B}$-inertly reach *SmallSp*. If $\mathit{SmallSp} = \varnothing$, then AvoidSml $= \varnothing$.

ReachAlw: states in B that can always $\mathcal{B}$-inertly reach both splitters *SmallSp* and *LargeSp* if both are non-empty. Otherwise, ReachAlw consists of states in B that can always $\mathcal{B}$-inertly reach the non-empty splitter. "Always ... reach" here also means: if a $\mathcal{B}$-inert transition starts in such a state, its target state, which is in B, must also be in ReachAlw.

NewBotSt: states in B that do not fit in any of the three sets above. Original bottom states cannot end up in NewBotSt, and new bottom states will always end up in this sub-block.

Figure 1 contains two examples of how a block B' is split. Here unlabelled transitions are $\mathcal{B}$-inert transitions in B'. The next lemma expresses that new bottom states end up in NewBotSt and Invariant 3.4 is preserved under the four-way-split.

▶ **Lemma 3.7.** Consider a partition $\mathcal{B}$ and $B \in \mathcal{B}$. Assume given $a \in \mathit{Act}$ and disjoint $\mathcal{B}_{\mathrm{sml}}, \mathcal{B}_{\mathrm{lrg}} \subseteq \mathcal{B}$ with $a \neq \tau$ or $B \notin \mathcal{B}_{\mathrm{sml}} \cup \mathcal{B}_{\mathrm{lrg}}$. Let

$$\mathit{SmallSp} := B \xrightarrow{a} \bigcup \mathcal{B}_{\mathrm{sml}} \qquad \text{and} \qquad \mathit{LargeSp} := B \xrightarrow{a} \bigcup \mathcal{B}_{\mathrm{lrg}}.$$

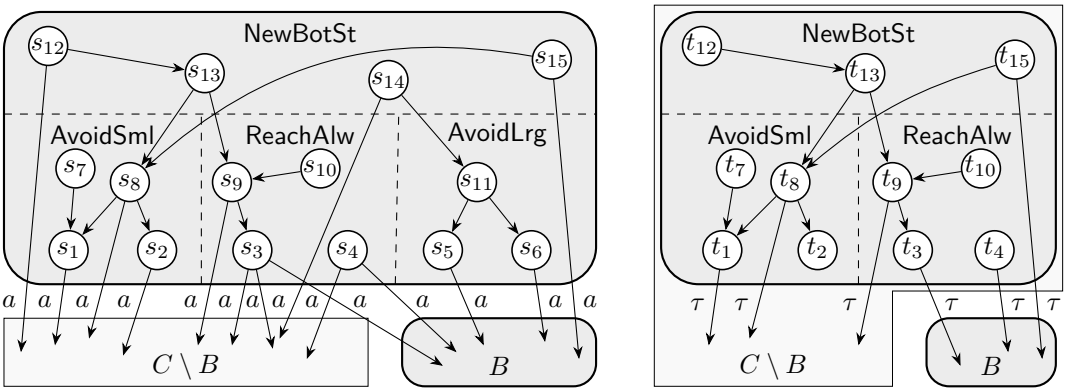

Figure 1 Left: four-way-split$\mathcal{B}(B', B' \xrightarrow{a} B, B' \xrightarrow{a} C \setminus B)$. Right: four-way-split$\mathcal{B}(B', B' \xrightarrow{\tau} B, \varnothing)$.

Assume further that $\mathit{SmallSp} \cup \mathit{LargeSp} \neq \varnothing$. Let $\mathcal{B}'$ be the partition $(\mathcal{B} \setminus \{B\}) \cup (\{\mathsf{AvoidLrg}, \mathsf{AvoidSml}, \mathsf{ReachAlw}, \mathsf{NewBotSt}\} \setminus \{\varnothing\})$ with the mentioned sets being the results of the call

four-way-split$\mathcal{B}(B, \mathit{SmallSp}, \mathit{LargeSp})$.

Then

1. For any state $s \in \mathsf{NewBotSt}$ it holds that $s \notin \mathit{Bottom}(B)$.
2. For any state $s \in \mathit{Bottom}(\mathsf{AvoidLrg}) \cup \mathit{Bottom}(\mathsf{AvoidSml}) \cup \mathit{Bottom}(\mathsf{ReachAlw})$ it holds that $s \in \mathit{Bottom}(B)$.
3. If Invariant 3.4 holds for $\mathcal{B}$, then Invariant 3.4 holds for $\mathcal{B}'$.

Proof.

1. We use contraposition. Assume that $s \in \mathit{Bottom}(B)$. Then s does not have an outgoing $\mathcal{B}$-inert transition. By construction s ends up in one of the sets AvoidSml, AvoidLrg and ReachAlw. Hence, $s \notin \mathsf{NewBotSt}$.
2. This follows by observing that if there is a bottom state $s \in B_s$ where B_s is any of the blocks AvoidLrg, AvoidSml and ReachAlw which was not a bottom state in B, then there is transition $s \xrightarrow{\tau} s'$ with $s' \in B$ but $s' \notin B_s$. Going through the various cases, it follows that s cannot be in any of the three sets B_s, and therefore must be part of NewBotSt.
3. Using that Invariant 3.4 is valid for $\mathcal{B}$, we only need to show that two states $s, t \in B$ that have been moved to different blocks in $\mathcal{B}'$ are not branching bisimilar.
   Assume $s \in \mathsf{ReachAlw}$ and $t \in \mathsf{AvoidSml}$. Then s can $\mathcal{B}$-inertly reach $\mathit{SmallSp}$. This means, using the invariant and the conditions of this lemma, it can reach a non-$\leftrightarrows_b$-inert transition in $\mathit{SmallSp}$. As $t \in \mathsf{AvoidSml}$, t cannot mimic this transition, and hence $s \not\leftrightarrows_b t$. More concretely, all states in ReachAlw and AvoidSml are not branching bisimilar. Similarly, one proves that the states in ReachAlw and AvoidLrg, respectively, the states in AvoidSml and AvoidLrg are not branching bisimilar.
   Now assume $s \in \mathsf{NewBotSt}$. We claim that s can $\mathcal{B}$-inertly reach both $\mathit{SmallSp}$ and $\mathit{LargeSp}$ unless they are empty, and s can $\mathcal{B}$-inertly reach AvoidSml or AvoidLrg. This can be seen as follows. If $s \in B$ can $\mathcal{B}$-inertly reach $\mathit{SmallSp}$ but not $\mathit{LargeSp}$ (and $\mathit{LargeSp} \neq \varnothing$), then $s \in \mathsf{AvoidLrg}$. Similarly, if $s \in B$ can $\mathcal{B}$-inertly reach $\mathit{LargeSp}$ but not $\mathit{SmallSp}$ (and $\mathit{SmallSp} \neq \varnothing$), then $s \in \mathsf{AvoidSml}$. So, s can reach all non-empty splitters. As $s \notin \mathsf{ReachAlw}$, it can $\mathcal{B}$-inertly reach a bottom state $s' \in \mathit{Bottom}(B) \setminus \mathsf{ReachAlw}$. As $s' \notin \mathsf{NewBotSt}$ (by dictum 1), we have $s' \in \mathsf{AvoidSml}$ or $s' \in \mathsf{AvoidLrg}$.
   If $t \in \mathsf{AvoidSml}$ or $t \in \mathsf{AvoidLrg}$, then s can $\mathcal{B}$-inertly reach a non-$\leftrightarrows_b$-inert transition in $\mathit{SmallSp}$ or $\mathit{LargeSp}$, respectively, that t cannot mimic.

Now consider $t \in \mathsf{ReachAlw}$. We know that s can $\mathcal{B}$-inertly reach a state $s' \in \mathsf{AvoidSml} \cup \mathsf{AvoidLrg}$, and as argued above $t \not\leftrightarrows_b s'$. Suppose we can find a state t' reachable with internal transitions from t that mimics the path from s to s' such that $t' \leftrightarrows_b s'$. Using Invariant 3.4 for $\mathcal{B}$, t' must be $\mathcal{B}$-inertly reachable and $t' \in B$. But then $t' \in \mathsf{ReachAlw}$ and $s' \not\leftrightarrows_b t'$. So, no such t' exists, and hence $s \not\leftrightarrows_b t$. ◀

The following lemma explains which calls to four-way-split$\mathcal{B}$ are needed to reestablish Invariant 3.5 after splitting a constellation $C \in \mathcal{C} \setminus \mathcal{B}$. For now we exclude blocks that contain new bottom states, i.e. sub-blocks of $\mathsf{NewBotSt}$, cf., Lemma 3.7.1 and 3.7.2.

▶ **Lemma 3.8.** Consider a partition $\mathcal{B}$ refining a partition of constellations $\mathcal{C}$ for which Invariant 3.5 holds. Let the refined set of constellations $\mathcal{C}'$ be $(\mathcal{C} \setminus \{C\}) \cup \{B, C \setminus B\}$ for some $B \in \mathcal{B}$ and $C \in \mathcal{C}$ and $B \subsetneq C$.

Let $\mathcal{B}'$ be the result of the iterative refinements $(\mathcal{B}' \setminus \{B'\}) \cup (\{\mathsf{AvoidLrg}, \mathsf{AvoidSml}, \mathsf{ReachAlw}, \mathsf{NewBotSt}\} \setminus \{\varnothing\})$ with the mentioned sets being the results of the following calls. Assume that for every action $a \in Act$ these calls take place for blocks $B' \in \mathcal{B}'$ satisfying the indicated conditions such that S is covered by these blocks B', blocks not satisfying the conditions, singleton blocks with $|B'| = 1$, or blocks that refine some block $\mathsf{NewBotSt}$:

four-way-split$\mathcal{B}(B', B' \xrightarrow{a} B, B' \xrightarrow{a} C \setminus B)$	if $a \neq \tau$ or $B' \not\subseteq C$, and there are $s_1, s_2 \in B'$ such that $s_1 \xrightarrow{a} t_1, s_2 \xrightarrow{a} t_2$ and $t_1 \in B, t_2 \in C \setminus B$,
four-way-split$\mathcal{B}(B', B' \xrightarrow{\tau} B, \varnothing)$	if $B' \subseteq C \setminus B$ and there is an $s \in B'$ such that $s \xrightarrow{\tau} t$ and $t \in B$,
four-way-split$\mathcal{B}(B', B' \xrightarrow{\tau} C \setminus B, \varnothing)$	if $B' \subseteq B$ and there is an $s \in B'$ such that $s \xrightarrow{\tau} t$ and $t \in C \setminus B$.

Then Invariant 3.5 holds for all blocks in $\mathcal{B}'$ (except for those that refine some block $\mathsf{NewBotSt}$) and $\mathcal{C}'$.

Proof. Consider a state $s \in \hat{B}$ for any block $\hat{B} \in \mathcal{B}'$ not refining any $\mathsf{NewBotSt}$. Assume that a non-$\mathcal{C}'$-inert transition $s \xrightarrow{a} t$ exists with $t \in \hat{C}$ for some constellation $\hat{C} \in \mathcal{C}'$. So, $a \neq \tau$ or $s \notin \hat{C}$. We must prove that any $s' \in \mathit{Bottom}(\hat{B})$ has a transition $s' \xrightarrow{a} t'$ with $t' \in \hat{C}$.

Let $\hat{B}'$ be the block $\hat{B} \subseteq \hat{B}' \in \mathcal{B}$. Note that $s' \in \mathit{Bottom}(\hat{B}')$ by Lemma 3.7.2.

Case $\hat{C} \not\subseteq C$: That means that $\hat{C} \in \mathcal{C}$. Using Invariant 3.5 (for $\hat{B}' \in \mathcal{B}$ and $\hat{C} \in \mathcal{C}$), s' has a transition $s' \xrightarrow{a} t'$ with $t' \in \hat{C}$.

Case $\hat{B} \subseteq C$, $a = \tau$ and $\hat{C} = B$: If $\hat{B} \subseteq B = \hat{B}'$, then $s \xrightarrow{\tau} t$ were $\mathcal{C}'$-inert. So, we can assume $\hat{B} \subseteq C \setminus B$. There is a block $\hat{B}''$ with $\hat{B} \subseteq \hat{B}'' \subseteq \hat{B}'$ such that a call

four-way-split$\mathcal{B}(\hat{B}'', \hat{B}'' \xrightarrow{\tau} B, \varnothing)$

must have taken place. In this case $\hat{B}$ is a sub-block of $\mathsf{ReachAlw}$ or $\mathsf{NewBotSt}$. In the last case there is nothing to check. But $\hat{B} \subseteq \mathsf{ReachAlw}$ implies that $s' \in \mathit{Bottom}(\hat{B})$ has a transition $s' \xrightarrow{\tau} t'$ with $t' \in B$, as had to be shown.

Case ($\hat{B} \not\subseteq C$ or $a \neq \tau$) and $\hat{C} = B$: If $(\hat{B} \xrightarrow{a} C \setminus B) = \varnothing$, we know by Invariant 3.5 (for $\hat{B}' \in \mathcal{B}$ and $C \in \mathcal{C}$) that $s' \in \mathit{Bottom}(\hat{B})$ has a transition $s' \xrightarrow{a} t'$ with $t' \in C$, and as the transition cannot go to $C \setminus B$, it must go to B, which we had to prove.

Otherwise, $(\hat{B} \xrightarrow{a} C \setminus B) \neq \varnothing$, and there is a block $\hat{B}''$ with $\hat{B} \subseteq \hat{B}'' \subseteq \hat{B}'$ for which a call

four-way-split$\mathcal{B}(\hat{B}'', \hat{B}'' \xrightarrow{a} B, \hat{B}'' \xrightarrow{a} C \setminus B)$

took place with $\hat{B}$ a sub-block of $\mathsf{AvoidLrg}$, $\mathsf{ReachAlw}$ or $\mathsf{NewBotSt}$. The last case does not lead to a proof obligation. But $\hat{B} \subseteq \mathsf{AvoidLrg}$ or $\hat{B} \subseteq \mathsf{ReachAlw}$ implies that $s' \in \mathit{Bottom}(\hat{B})$ has a transition $s' \xrightarrow{a} t'$ with $t' \in B$, as had to be shown.

Algorithm 1 Branching bisimulation partitioning.

1.1:	Find τ-SCCs and contract each of them to a single state	$O(\lvert Act\rvert + m)$
1.2:	$\mathcal{C} := \{S\}$; $\mathcal{B} := \{S\}$; $has\_new\_bottom\_states(S) := true$	
1.3:	Call stabilise$\mathcal{B}$() to create the initial partition	
1.4:	**while** a $B \in \mathcal{B}$ exists such that $B \notin \mathcal{C}$ **do**	$\leq n$ iterations
1.5:	Select some $B \in \mathcal{B}$ and $C \in \mathcal{C}$ such that $B \subseteq C$ and $\lvert B\rvert \leq \frac{1}{2}\lvert C\rvert$	
1.6:	$\mathcal{C} := (\mathcal{C} \setminus \{C\}) \cup \{B, C \setminus B\}$ and maintain data structures accordingly	
1.7:	$\mathcal{M} := \{s \xrightarrow{a} t \mid s \in S, a \in Act, t \in B,$ and $a \neq \tau$ or $s \notin B\}$	$O(\lvert in(B)\rvert)$
1.8:	Split B under the outgoing transitions that have just become non-$\mathcal{C}$-inert by calling four-way-split$\mathcal{B}(B, B \xrightarrow{\tau} (C \setminus B), \varnothing)$	+ split
1.9:	**while** $\mathcal{M} \neq \emptyset$ **do**	$\leq \lvert in(B)\rvert$ iter'ns
1.10:	Pick some BLC set $SmallSp := (B' \xrightarrow{a} B) \subseteq \mathcal{M}$ for some block B'	
1.11:	$\mathcal{M} := \mathcal{M} \setminus SmallSp$	
1.12:	**if** $\lvert B'\rvert > 1 \wedge \neg has\_new\_bottom\_states(B')$ **then**	
1.13:	**if** $B' \subseteq C \wedge a = \tau$ **then**	
1.14:	Split B' under the transitions that have become non-$\mathcal{C}$-inert by calling four-way-split$\mathcal{B}(B', SmallSp, \varnothing)$	$O(1)$ + split
1.15:	**else**	
1.16:	$LargeSp := B' \xrightarrow{a} (C \setminus B)$	
1.17:	**if** $LargeSp \neq \varnothing$ **then**	
1.18:	Split B' by calling four-way-split$\mathcal{B}(B', SmallSp, LargeSp)$	
1.19:	Call stabilise$\mathcal{B}$() to stabilise blocks with new bottom states	
1.20:	**return** $\mathcal{B}$	

Case $\hat{B} \subseteq C$, $a = \tau$ and $\hat{C} = C \setminus B$: If $\hat{B} \subseteq C \setminus B$, then $s \xrightarrow{\tau} t$ were $\mathcal{C}'$-inert. So we can assume $\hat{B} \subseteq B = \hat{B}'$. There is a block $\hat{B}''$ with $\hat{B} \subseteq \hat{B}'' \subseteq B$ such that a call

four-way-split$\mathcal{B}(\hat{B}'', \hat{B}'' \xrightarrow{\tau} C \setminus B, \varnothing)$

must have taken place, where $\hat{B}$ is a sub-block of ReachAlw or NewBotSt. The latter case is not interesting. Again, $\hat{B} \subseteq$ ReachAlw implies that $s' \in Bottom(\hat{B})$ has a transition $s' \xrightarrow{\tau} t'$ with $t' \in C \setminus B$, as had to be shown.

Case ($\hat{B} \not\subseteq C$ or $a = \tau$) and $\hat{C} = C \setminus B$: If $(\hat{B} \xrightarrow{a} B) = \varnothing$, we know by Invariant 3.5 (for $\hat{B}' \in \mathcal{B}$ and $C \in \mathcal{C}$) that $s' \in Bottom(\hat{B})$ has a transition $s' \xrightarrow{a} t'$ with $t' \in C$, and as the transition cannot go to B, it must go to $C \setminus B$, which we had to prove.
Otherwise, $(\hat{B} \xrightarrow{a} B) \neq \varnothing$, and there is a block $\hat{B}''$ such that $\hat{B} \subseteq \hat{B}'' \subseteq \hat{B}'$ and a call

four-way-split$\mathcal{B}(\hat{B}'', \hat{B}'' \xrightarrow{a} B, \hat{B}'' \xrightarrow{a} C \setminus B)$

must have taken place, and $\hat{B}$ is a subset of AvoidSml or ReachAlw or NewBotSt. We can ignore the latter case. But $\hat{B} \subseteq$ AvoidSml or $\hat{B} \subseteq$ ReachAlw implies that $s' \in Bottom(\hat{B})$ has a transition $s' \xrightarrow{a} t'$ with $t' \in C \setminus B$, as had to be shown. ◀

Blocks with new bottom states, which were excluded by Lemma 3.8, are handled by the function stabilise$\mathcal{B}$. It assumes that blocks without new bottom states are already stable and stabilises the blocks with new bottom states, which concretely means that it refines them under all their outgoing non-$\mathcal{C}$-inert transitions such that for the resulting partition Invariant 3.5 holds, while maintaining Invariant 3.4.

Main Algorithm. We are now in the situation to explain Algorithm 1. First we need to take care that the Invariants 3.4 and 3.5 become valid. Therefore, in line 1.2 we set the partition $\mathcal{B}$ and the set of constellations $\mathcal{C}$ to $\{S\}$. This makes Invariant 3.4 trivially true.

If a block B has new bottom states, we use the predicate $has\_new\_bottom\_states(B)$ to indicate so. This is not only the case when block B is a refinement of some block NewBotSt but it also applies to the initial block, and this is explicitly indicated in line 1.2. Invariant 3.5 is now made valid by one call to stabilise$\mathcal{B}$ in line 1.3. Concretely, this call splits $\{S\}$ into a partition $\mathcal{B}$ such that all bottom states in each block B in $\mathcal{B}$ have exactly the same outgoing non-$\mathcal{B}$-inert actions, while keeping branching bisimilar states in the same block.

As stated above, the purpose of the main algorithm is to refine $\mathcal{C}$ such that it ultimately coincides with $\mathcal{B}$. This happens in the loop starting at line 1.4. If $\mathcal{B}$ and $\mathcal{C}$ do not coincide, there is at least one block $B \subseteq C$ with size smaller than or equal to $\frac{1}{2}|C|$. We replace the constellation C in $\mathcal{C}$ by two new constellations B and $C \setminus B$ (line 1.6).

Our main task is to make Invariant 3.5 valid again. This is done by taking care that all calls to four-way-split$\mathcal{B}$ as required by Lemma 3.8 are performed. This guarantees that Invariant 3.5 is valid except for blocks of the form NewBotSt. Therefore, one extra call is needed to stabilise$\mathcal{B}$ in line 1.19.

In order to effectively carry out the four-way-splits, transitions are organised in *block-label-constellation sets (BLC sets)*. Every BLC set contains the transitions $B \xrightarrow{a} C$ for a specific block $B \in \mathcal{B}$, action $a \in Act$, and constellation $C \in \mathcal{C}$. Whenever a constellation or a block is split (lines 1.6 and 2.45) these BLC sets are updated.

In line 1.8 there is a call to four-way-split$\mathcal{B}(B, B \xrightarrow{\tau} (C \setminus B), \varnothing)$ that corresponds with the third case in Lemma 3.8. So, that case has been covered.

For the remainder we put all non-$\mathcal{C}$-inert transitions to block B in a set $\mathcal{M}$. By traversing the BLC sets in $\mathcal{M}$ the other required calls to four-way split are done. In line 1.5 such a BLC set $SmallSp = B' \xrightarrow{a} B$ for some block B' is selected. For the complexity, it is important that $|B| < \frac{1}{2}|C|$ as this implies that the selected BLC set is small, hence its name.

If B' has only one state or is a refinement of some block NewBotSt and hence has new bottom states, according to Lemma 3.8 it does not need to be considered to be split (line 1.12).

Otherwise, there are two situations. The first one is when $a = \tau$ and B' was part of the constellation C (line 1.13). As $\mathcal{M}$ does not contain τ-transitions from B to B, this means $B' \in C \setminus B$ and the required second call four-way-split$\mathcal{B}(B', SmallSp, \varnothing)$ in Lemma 3.8 is made. The other situation, i.e., the **else** part at lines 1.15–1.18, exactly satisfies the conditions of the first mentioned call to four-way-split in Lemma 3.8. As the algorithm traverses all BLC sets that are part of $\mathcal{M}$, the whole set S is covered as required in Lemma 3.8.

From Lemma 3.8 it follows that at line 1.19 of the algorithm, Invariant 3.5 holds for all blocks, except for those that are of the shape NewBotSt. As stated above one call to stabilise$\mathcal{B}$ is sufficient to restore Invariant 3.5 for all blocks. Just for the record, Invariant 3.4 remains valid, as each call to four-way-split$\mathcal{B}$ preserves it (see Lemma 3.7.3), and, as explained in Section 5, stabilise$\mathcal{B}$ refines $\mathcal{B}$ only using four-way-split$\mathcal{B}$ also.

This means that at line 1.4 of Algorithm 1 both invariants hold. If the condition of the **while** is not valid, the partitions $\mathcal{B}$ and $\mathcal{C}$ are equal, and as argued above, these resulting partitions coincide exactly with branching bisimulation. As the initial set of states S is finite, the partition $\mathcal{C}$ can only be refined finitely often, which happens in the body of the while loop. This means that this algorithm always terminates with the required answer.

4 Four-Way Split

This section explains how four-way-split$\mathcal{B}(B, SmallSp, LargeSp)$ refines block B under two splitters, i.e., sets of transitions *SmallSp* and *LargeSp*. The sets contain all transitions from B with a given label a to one or two given constellations.

Algorithm 2 Refine a block under one or two sets of transitions.

2.1: **function** four-way-split$_{\mathcal{B}}$(block B, small splitter $SmallSp$, large splitter $LargeSp$)
2.2: ReachAlw := $\{s \in Bottom(B) \mid (SmallSp = \varnothing \vee \exists s \xrightarrow{a} t \in SmallSp) \wedge (LargeSp = \varnothing \vee \exists s \xrightarrow{a} t \in LargeSp)\}$
2.3: AvoidLrg := $\{s \in Bottom(B) \mid (SmallSp = \varnothing \vee \exists s \xrightarrow{a} t \in SmallSp) \wedge \neg(LargeSp = \varnothing \vee \exists s \xrightarrow{a} t \in LargeSp)\}$
2.4: AvoidSml := $Bottom(B) \setminus$ (ReachAlw $\cup$ AvoidLrg)
2.5: **if** AvoidLrg $= \varnothing \wedge$ AvoidSml $= \varnothing$ **then**
2.6: **return** // B is stable under $SmallSp$ and $LargeSp$.
2.7: pot-ReachAlw := $\{s \in B \setminus Bottom(B) \mid \exists s \xrightarrow{a} t \in SmallSp \wedge (LargeSp = \varnothing \vee \exists s \xrightarrow{a} t \in LargeSp)\}$
2.8: HitSmall := $\{s \in B \setminus Bottom(B) \mid \exists s \xrightarrow{a} t \in SmallSp \wedge \neg(LargeSp = \varnothing \vee \exists s \xrightarrow{a} t \in LargeSp)\}$
2.9: **for all** $s \in$ pot-ReachAlw **do**
2.10: $s.count := |s \xrightarrow{\tau} B|$
2.11: NewBotSt := $\varnothing$; pot-AvoidLrg := $\varnothing$; pot-AvoidSml := $\varnothing$

$O(|SmallSp|)$ or $O(|Bottom(B) \xrightarrow{a} S \cap LargeSp|)$

2.12: **begin coroutines**
2.13: If some subset becomes too large at any time, abort its coroutine immediately.

cor $R =$ ReachAlw, AvoidSml, AvoidLrg **do**
2.14: **for all** $s \in R$ **do**
2.15: **for all** $t \xrightarrow{\tau} s \in in(s)$ **do**
2.16: **if** $t \notin B \setminus$ NewBotSt **then**
2.17: Skip $t \xrightarrow{\tau} s$ (go to line 2.16)
2.18: **if** $t \notin$ pot-R **then**
2.19: **if** $R =$ AvoidSml $\wedge\, t \in$ HitSmall $\vee\, \exists R'.t \in$ pot-R' **then**
2.20: Add t to NewBotSt
2.21: Skip $t \xrightarrow{\tau} s$ (go to line 2.16)
2.22: Add t to pot-R
2.23: $t.count := |t \xrightarrow{\tau} B|$
2.24: $t.count := t.count - 1$
2.25: **if** $t.count \neq 0$ **then**
2.26: Skip $t \xrightarrow{\tau} s$ (go to line 2.16)
2.27: **if** $R =$ AvoidLrg **then**
2.28: **for all** $t \xrightarrow{b} u \in out(t)$ **do**
2.29: **if** $t \xrightarrow{b} u \in LargeSp$ **then**
2.30: Add t to NewBotSt
2.31: Skip $t \xrightarrow{\tau} s$ (go to line 2.16)
2.32: Add t to R
2.33: Declare R finished
2.34: **if** 3 coroutines have finished **then**
2.35: Terminate the coroutines
2.36: NewBotSt := NewBotSt $\cup$ (pot-$R \setminus R$)
2.37: **if** ReachAlw&AvoidLrg are finished **then**
2.38: NewBotSt := NewBotSt $\cup$ HitSmall $\setminus$ (ReachAlw $\cup$ AvoidLrg)

$O(|in(R)|)$ or $O(|out(\text{AvoidLrg})| + |out(Bottom(\text{NewBotSt}))|)$ $O(|in(R)|)$ $O(|SmallSp|)$

loop forever
for all unexplored $s \in$ NewBotSt **do**
for all $t \xrightarrow{\tau} s \in in(s)$ **do**
if $t \in B$ **then**
Add t to NewBotSt
if 2 coroutines have finished **then**
if $LargeSp$ is fully explored $\vee$ AvoidLrg is finished **then**
Declare NewBotSt finished
Terminate the coroutines
Pick unexplored $t \xrightarrow{a} s \in LargeSp$
if $t \notin$ ReachAlw $\cup$ AvoidSml $\wedge\, t \notin Bottom(B)$ **then**
Add t to NewBotSt

$O(|in(R)|)$ or $O(|in(\text{NewBotSt})| + |out(\text{NewBotSt})| + |out(finished)|)$

2.39: **end coroutines**
2.40: **if** NewBotSt is unfinished $\wedge$ ReachAlw $\cup$ AvoidLrg $\cup$ AvoidSml $= B$ **then**
2.41: Declare NewBotSt finished and the largest subset aborted
2.42: Create sub-blocks for the finished, non-empty subsets
2.43: Update transitions that have become non-$\mathcal{B}$-inert and BLC sets
2.44: **if** NewBotSt $\neq \varnothing$ **then**
2.45: *has_new_bottom_states*(NewBotSt) := *true*
2.46: **return**

$O(|in(finished)| + |out(finished)|)$

Splitting block B must be done in a time proportional to the size of *SmallSp* and the sizes of all but the largest block into which B is split. The principle "process the smaller half" then implies that every state and transition is only visited $O(\log n)$ times. However, we do not know a priori which of the three parts are the smallest. Therefore, we start four coroutines for the four parts, run them in lockstep and abort the coroutine that belongs to the largest part. This ensures that time proportional to the three smallest parts is used.

Preconditions. At least one splitter must be non-empty. If both splitters are given (as in line 1.18 of Algorithm 1), it is known that every bottom state of B has a transition in at least one of the two splitters. We use data structures such that it is possible to quickly determine whether states with a transition in *SmallSp* also have a transition in *LargeSp*.

We now go through the pseudocode shown as Algorithm 2.

Preparation: Splitting the Bottom States. In lines 2.2–2.11, the algorithm initialises a number of sets of states, mainly to decide for every bottom state to which part it belongs.

In lines 2.2–2.4 in case $\mathit{SmallSp} \neq \varnothing$ every transition in *SmallSp* is visited, and its source state is moved to ReachAlw or AvoidLrg, depending on whether it has a transition in *LargeSp* or not. Bottom states that have not been moved belong to AvoidSml. In case $\mathit{SmallSp} = \varnothing$, the transitions in *LargeSp* that start in a bottom state are traversed, to move these bottom states to ReachAlw. The states that have not been moved belong to AvoidLrg.

If some non-bottom state has a transition in *SmallSp*, it is not immediately known to which part it belongs; it might happen that it has a $\mathcal{B}$-inert transition to another part. If the non-bottom state has a transition in the small splitter but not in $\mathit{LargeSp} \neq \varnothing$ (like s_{15} in Figure 1 at the left), it may end up in any part except AvoidSml. To register this such a state is temporarily added to HitSmall in line 2.8. It will be processed further in line 2.20 left.

If the non-bottom state has a transition in all (non-empty) splitters, it is potentially in ReachAlw, unless it has a $\mathcal{B}$-inert transition to one of the other sub-blocks. In the latter case, it will be in NewBotSt. We add it temporarily to pot-ReachAlw to register this in line 2.7.

If all bottom states belong to ReachAlw, the split is trivial (lines 2.5–2.6). Even if all bottom states belong to one of AvoidLrg or AvoidSml, there still can be some non-bottom states with transitions causing a split. Such states will be moved to NewBotSt.

Coroutines: Extending the Splits to Non-bottom States. After all bottom states have been distributed over the initial sets, we have to extend the distribution to the non-bottom states. To stay within the time bounds, we need to find all the states in the three smallest parts in time proportional to the number of their incoming and outgoing transitions.

Of the four coroutines, three (for ReachAlw, AvoidSml and AvoidLrg) are almost the same. Their code is shown on the left of lines 2.12–2.41, where we use the keyword **cor** to indicate the three coroutines. The basic idea is the following. If all outgoing $\mathcal{B}$-inert transitions of some non-bottom state go to the same set, the state also belongs to this set. To avoid checking any transition more than once, we go through all incoming $\mathcal{B}$-inert transitions of the respective set and count how many outgoing $\mathcal{B}$-inert transitions of its source are not yet known to point to this set. If the $\mathcal{B}$-inert transitions point to different sets, the state belongs to NewBotSt (line 2.21 left).

If the number of unknown transitions is 0, the source is added to the respective set, unless it has an incompatible transition in a splitter: states with a transition in both splitters can only be in ReachAlw or NewBotSt and were therefore already added to pot-ReachAlw; this ensures that they move to NewBotSt as soon as it is found out that they have a transition to

Algorithm 3 Stabilise the partition with respect to new bottom states.

3.1: **function** stabilise$\mathcal{B}$()
3.2: $\hat{\mathcal{Q}} := \varnothing$
3.3: **for all** blocks B with $has\_new\_bottom\_states(B)$ **do**
3.4: $\quad \hat{\mathcal{Q}} := \hat{\mathcal{Q}} \cup \{s \xrightarrow{a} t \mid s \in B \wedge (a \neq \tau \vee constellation(s) \neq constellation(t))\}$ } $O(|out(Bottom^+(B))|)$
3.5: $\quad$ **for all** $s \in Bottom(B)$ **do**
3.6: $\quad\quad$ Move the non-$\mathcal{C}$-inert transitions out of s to the front of their BLC sets } $O(|out(Bottom(B))|)$
3.7: $\quad has\_new\_bottom\_states(B) := false$
3.8: **while** $\hat{\mathcal{Q}} \neq \varnothing$ **do** } $\leq |out(Bottom^+(B))|$ iter'ns
3.9: $\quad$ Pick some BLC set $Sp := (B \xrightarrow{a} C) \subseteq \hat{\mathcal{Q}}$
3.10: $\quad \hat{\mathcal{Q}} := \hat{\mathcal{Q}} \setminus Sp$
3.11: $\quad$ **if** $|B| > 1$ **then** } $O(1)$ + split
3.12: $\quad\quad$ Split B under Sp by calling four-way-split$\mathcal{B}(B, \varnothing, Sp)$
3.13: $\quad\quad$ **if** the split produced some NewBotSt $\neq \varnothing$ **then**
3.14: $\quad\quad\quad \hat{\mathcal{Q}} := \hat{\mathcal{Q}} \cup \{s \xrightarrow{a} t \mid (s \xrightarrow{a} t) \notin \hat{\mathcal{Q}} \wedge s \in \text{NewBotSt} \wedge (a \neq \tau \vee constellation(s) \neq constellation(t))\}$ } $O\left(|out(Bottom^+(\text{NewBotSt})) \setminus \hat{\mathcal{Q}}|\right)$
3.15: $\quad\quad\quad$ **for all** $s \in Bottom(\text{NewBotSt})$ **do**
3.16: $\quad\quad\quad\quad$ Move the non-$\mathcal{C}$-inert transitions out of s to the front of their BLC sets } $O\left(|out(Bottom(\text{NewBotSt}))|\right)$
3.17: $\quad\quad\quad has\_new\_bottom\_states(\text{NewBotSt}) := false$
3.18: **return**

AvoidSml or AvoidLrg. – States with a transition in *SmallSp* but not in *LargeSp* cannot be in AvoidSml. If they would be added to that set, the test in line 2.20 left ensures that they will be added to NewBotSt instead. – States with a transition in *LargeSp* but not in *SmallSp* cannot be in AvoidLrg. But as the algorithm is not allowed to go through all transitions in *LargeSp* a priori, they are more difficult to handle. Only if the number of unknown transitions is 0 and the state is still a candidate for AvoidLrg, we are allowed to spend the time to check whether it has a transition in *LargeSp* by looking through its non-$\mathcal{C}$-inert outgoing transitions in lines 2.29–2.33 left. Section 6 explains why this fits the time bounds.

When all incoming $\mathcal{B}$-inert transitions of a set have been visited, no more states can be added, so that set is finished. The states that had some $\mathcal{B}$-inert transitions to the set but still have unknown transitions must have transitions to some other set as well, so they are added to NewBotSt in line 2.38 left. If AvoidSml is the only coroutine that still runs on the left side, any unassigned states remaining in HitSmall also belong to NewBotSt, as they cannot be in AvoidSml (see lines 2.39–2.40 left).

The fourth coroutine, the one for NewBotSt, follows a slightly simpler principle. Every $\mathcal{B}$-inert predecessor of a NewBotSt-state is also in NewBotSt, independent of other transitions it might have. This happens in line 2.18 right. The other three coroutines may also occasionally add states to NewBotSt and therefore, different from the first three coroutines, we cannot end the coroutine as soon as all incoming $\mathcal{B}$-inert transitions of the current NewBotSt have been visited. Instead, we wait in an (outer) infinite loop. The outer loop can terminate when at least AvoidLrg and one other coroutine have finished.

If only the coroutines for AvoidLrg and NewBotSt are still running, the latter will go through *LargeSp* to find states that cannot be in AvoidLrg in lines 2.24–2.27 right. If the source state of some transition in *LargeSp* has not yet been inserted into a sub-block, it cannot be in AvoidLrg, so it must be in NewBotSt.

5 Stabilising New Bottom States

In this section we explain stabilise$\mathcal{B}$ given in Algorithm 3. The procedure stabilise$\mathcal{B}$ goes through all blocks with new bottom states and completely stabilises them.

Algorithm 3 shows the pseudocode of the stabilisation. In lines 3.3–3.7, all transitions out of the states in blocks with new bottom states are collected in $\hat{\mathcal{Q}}$. Additionally, all non-$\mathcal{C}$-inert transitions out of bottom states, which are all new, are put to the front of their BLC sets. In line 3.12 there is a call to four-way-split$\mathcal{B}$ that only traverses these transitions in front of the BLC set to determine whether B needs to be split.

In lines 3.8–3.12, each BLC set is iteratively chosen as potential splitter Sp and its source block B is stabilised under it. In the preparation phase of four-way-split$\mathcal{B}$, we can now visit the transitions in Sp starting in bottom states without spending time to pick them from between those starting in non-bottom states. If this refinement leads to further new bottom states, the sub-block NewBotSt is not empty. This block with new bottom states is immediately added to $\hat{\mathcal{Q}}$ as it requires to be stabilised again, see lines 3.13–3.17.

To ensure that every transition out of a new bottom state is visited only a bounded number of times, we have to take into account that some transitions out of the set NewBotSt may already be in $\hat{\mathcal{Q}}$ and do not need to be added a second time.

6 Complexity

We attribute computation steps to certain transitions or states in the LTS to account for the time bound. There are three kinds of timing attributions:

C. When a constellation C is split into B and $C \setminus B$, time is spent proportionally to the states and incoming and outgoing transitions of B.

B. When a block B is split into AvoidLrg, AvoidSml, ReachAlw and NewBotSt, time is spent proportionally to the states and incoming and outgoing transitions of the three smallest sub-blocks.

N. When a new bottom state is found, time is spent proportionally to this state and its outgoing transitions.

We first formulate three lemmas to distribute all execution steps (except the initialisation) over the three cases above, and then state the main timing result.

▶ **Lemma 6.1.** Every step in four-way-split$\mathcal{B}$ (Algorithm 2) falls under one of the timing items C, B, or N above.

Proof. We can ignore the coroutine for the largest sub-block because it is aborted as soon as it is clearly the largest (line 2.13) or the three other coroutines are finished. The time it spends is at most proportional to the time attribution of the second-largest coroutine.

Initialisation (lines 2.2–2.11): If the function is called from the main algorithm, *SmallSp* either consists of outgoing transitions of B (line 1.8) or of incoming transitions of B (lines 1.14 and 1.18). Then the sets ReachAlw, AvoidLrg, AvoidSml, pot-ReachAlw and HitSmall can be initialised by going through the transitions in *SmallSp*, which falls under timing item C. Note that we require that for states with a transition in *SmallSp*, one can determine in time $O(1)$ whether they have a transition in *LargeSp*. This allows to distinguish ReachAlw from AvoidLrg and pot-ReachAlw from HitSmall.

If the function is called from stabilise$\mathcal{B}$ (line 3.12), $SmallSp = \varnothing$, and *LargeSp* consists of outgoing transitions of a block with new bottom states. Then the sets ReachAlw and AvoidLrg can be distinguished by going through those transitions in *LargeSp* that start in

(new) bottom states, which falls under timing item N. The other sets in the initialisation are empty. Note that we only spend time on transitions in *LargeSp* starting in *bottom* states because we singled them out in lines 3.6 and 3.16.

All coroutines: Every coroutine generally runs a loop over all states in the respective sub-block and its incoming transitions. This can immediately be attributed to timing item B.

Coroutine for AvoidLrg (lines 2.29–2.33 left): The coroutine for AvoidLrg, once it has found a state t with $t.count = 0$, still needs to check whether t has a transition in *LargeSp* that could not be handled during initialisation. Depending on the outcome of this check, we attribute the time spent on the loop in lines 2.29–2.33 left differently:

- If t has no transition in *LargeSp*, we conclude that $t \in$ AvoidLrg, and we can attribute the time to the outgoing transitions of AvoidLrg, i.e. to timing item B.
- If, however, t has a transition in *LargeSp*, it is added to NewBotSt in line 2.32 left, and it becomes a new bottom state: all its outgoing $\mathcal{B}$-inert transitions point to another sub-block (namely AvoidLrg). Therefore we can attribute this to timing item N.

Finalising a sub-block $R =$ ReachAlw, AvoidSml or AvoidLrg: In line 2.38 left, states in pot-$R \setminus R$ are moved to NewBotSt. Every such state has a transition to R, so this is attributed to timing item B. In lines 2.39–2.40 left, the HitSmall-states are moved to NewBotSt. Because $|\text{HitSmall}| \leq |\mathit{SmallSp}|$, we can attribute this to timing item C.

Coroutine for NewBotSt: This coroutine has an outer infinite loop in line 2.14 right, when all states that are known to be in NewBotSt are already explored and the coroutine has to wait until a different coroutine adds another state to NewBotSt. Note that this will only happen as long as at least two coroutines different from NewBotSt are running. So, we can attribute the waiting time to the smaller of the two sub-blocks. When only the coroutines for AvoidLrg and NewBotSt are unfinished, the latter goes through the transitions in *LargeSp* in lines 2.24–2.27 right. Source states of these transitions cannot be in AvoidLrg, so they either are in a finished sub-block (and then that sub-block is not the largest one, so we are allowed to spend time on its outgoing transitions under timing item B) or they are in NewBotSt (and then they also fall under timing item B).

Final administration (lines 2.42–2.46): This can be organised in a way that only the states in the three smallest sub-blocks and their outgoing transitions are moved to new blocks and new BLC sets, respectively. So, we attribute it to timing item B. ◀

▶ **Lemma 6.2.** Every step in stabilise$\mathcal{B}$ (Algorithm 3) falls under one of the timing items C, B, or N above.

Proof. It can happen that we have to stabilise a block with new bottom states under BLC sets whose transitions all start in non-bottom states, but then *after* the respective call to four-way-split$\mathcal{B}$, this BLC set will contain a transition that starts in a new bottom state. We denote the set of the new bottom states together with these prospective new bottom states by "$\mathit{Bottom}^+(B)$".

Initialisation (lines 3.3–3.7): If one adds transitions to $\hat{\mathcal{Q}}$ in constant time per BLC set in line 3.4, then time $O(|\mathit{Bottom}^+(B)|)$ is spent. Lines 3.5–3.7 access new bottom states and their outgoing transitions. All these steps can be attributed to timing item N.

Main loop (lines 3.8–3.12): In every iteration one BLC set in $\hat{\mathcal{Q}}$ is handled, so there cannot be more iterations than transitions starting in $\mathit{Bottom}^+(B)$ (summed over all blocks B with new bottom states). The call to four-way-split$\mathcal{B}$ is attributed according to Lemma 6.1.

Adding another new bottom block (lines 3.13–3.17): This is the same operation as the loop body for the initialisation (lines 3.4–3.7), so we can attribute the timing in the same way. When adding new transitions to $\hat{\mathcal{Q}}$, one only has to take care not to spend time on BLC sets that have been added already earlier. ◀

▶ **Lemma 6.3.** Every step in the main loop in Algorithm 1 (lines 1.4–1.19) falls under one of the timing items C, B, or N above.

Proof. Splitting a constellation (lines 1.4–1.8): This part of the algorithm handles the states and transitions of B and is attributed to timing item C.

Stabilising under B and $C \setminus B$ (lines 1.9–1.19): Except for the calls to four-way-split$\mathcal{B}$ and stabilise$\mathcal{B}$ (which fall under Lemmas 6.1 and 6.2), these lines use constant time per iteration. Because $\mathcal{M} \subseteq in(B)$, there are not too many iterations. ◀

▶ **Theorem 6.4.** Under the reasonable assumption $|Act| \in O(m)$ and $n \in O(m)$, the running time of Algorithm 1 including its subroutines is in $O(m \log n)$.

Proof. Initialisation. The calculation of the τ-SCCs (line 1.1) takes $O(m)$ time via a standard algorithm [13]. Line 1.2 includes constructing the initial BLC sets. The essential step is to (bucket-)sort the transitions according to actions in time $O(|Act| + m) = O(m)$.

Steps under timing item C. State s can be in the small new constellation B (line 1.6) at most $\log_2 n$ times. Whenever this happens, time in $O(1 + |in(s)| + |out(s)|)$ is spent. Summing over all states, the total running time under timing item C is in $O(m \log n)$.

Steps under timing item B. State s can be in a small sub-block (when calling four-way-split$\mathcal{B}$) at most $\log_2 n$ times. Whenever this happens, time in $O(1 + |in(s)| + |out(s)|)$ is spent. Summing over all states, the total running time under timing item B is in $O(m \log n)$.

Steps under timing item N. State s is treated as new bottom state at most once during the whole algorithm. When this happens, time in $O(1 + |out(s)|)$ is spent. Summing over all states, the total running time under timing item N is in $O(m)$.

As now every part of the algorithm is accounted for, either directly or through one of the above lemmas, the result follows. ◀

Data Structures that Satisfy the Timing Constraints. We believe that many data structures are straightforward to implement, and we only describe the less obvious data structures here.

Transitions are stored in three orderings: incoming transitions per state (used e.g. in lines 1.6–1.7 and 2.16), outgoing transitions per state (used e.g. in lines 2.30 left and 3.6), and transitions per BLC set (used e.g. in lines 2.24 right and 3.4). The outgoing transitions are grouped per label and target constellation, and every block has a list of BLC sets with transitions that start in this block.

When splitting a constellation (line 1.6), the groups for outgoing transitions per state to C are split into two, such that the groups for *SmallSp* and *LargeSp* are next to each other. This allows to find out quickly whether a state with an outgoing transition in one of them also has a transition in the other (used to distinguish ReachAlw from AvoidLrg and pot-ReachAlw from HitSmall in lines 2.2–2.8). Similarly, the list entries for BLC sets containing transitions to C are split into two adjacent ones, and line 1.16 can find *LargeSp* (given *SmallSp*) in time $O(1)$.

During stabilise$\mathcal{B}$, we place BLC sets that are in $\hat{\mathcal{Q}}$ near the end of the list, so that line 3.14 only spends time on BLC sets that are currently not in $\hat{\mathcal{Q}}$.

To be absolutely sure that our correctness and timing analysis, as well as the implementation are correct we instrumented the implementation with both assertions and counters counting each operation on states and transitions and ran it on a large number of transition systems, most of which were randomly generated. Besides the invariants and the overall time bounds, it is checked that after the coroutines are finished, the counters for the largest sub-block have not been increased more often than the others, and also that the coroutine for NewBotSt did not wait too often without doing assignable work.

Table 1 Branching bisimulation benchmarks. – Running times are rounded to significant digits; trailing zeroes without following decimal point are insignificant. Bold times indicate the algorithm that is faster with 95% confidence.

	After SCC		After reduction		JGKW		This paper	
	#states	#trans	#states	#trans	time	memory	time	memory
vasy_574_13561	.57M	1.4M	3,577	16,168	20 s	1.3 G	**10 s**	**.91G**
vasy_6020_19353	1.4 M	3.9M	256	510	**1.7 s**	.83G	3. s	.83G
vasy_1112_5290	1.1 M	5.3M	265	1,300	11. s	.56G	**6. s**	**.41G**
cwi_2165_8723	.22M	8.7M	4,256	20,880	20 s	.95G	**10 s**	**.71G**
vasy_6120_11031	.61M	11 M	2,505	5,358	20 s	1.4 G	20 s	**1.1 G**
vasy_2581_11442	2.6 M	11 M	704,737	3,972,600	30 s	**1.4 G**	**20 s**	1.5 G
vasy_4220_13944	4.2 M	14 M	1,186,266	6,863,329	40 s	2.1 G	**20 s**	**1.4 G**
vasy_4338_15666	4.3 M	16 M	704,737	3,972,600	50 s	1.8 G	**30 s**	**1.8 G**
cwi_2416_17605	2.2 M	16 M	730	2,899	**9. s**	1.6 G	20 s	**1.1 G**
vasy_11026_24660	11 M	25 M	775,618	2,454,834	80 s	2.9 G	**60 s**	**2.6 G**
vasy_12323_27667	12 M	28 M	876,944	2,780,022	90 s	3.3 G	**70 s**	**3.0 G**
vasy_8082_42933	8.1 M	42 M	290	680	90 s	4.5 G	**50 s**	**3.3 G**
cwi_7838_59101	7.8 M	58 M	62,031	470,230	200 s	5.9 G	**120 s**	**4.2 G**
1394-fin-vvlarge	38 M	85 M	607,942	1,590,210	300 s	10 G	**200 s**	**8.1 G**
cwi_33949_165318	34 M	165 M	12,463	71,466	500 s	1.8 G	500 s	**1.3 G**

7 Benchmarks

In Table 1 we show some benchmarks illustrating the performance of the new algorithm. The benchmarks are mainly taken from the VLTS benchmark set[1]. We show the number of states and transitions after removal of τ-cycles under the header "After SCC". In the next columns the number of states after branching bisimulation reduction are given. Subsequently, the time and memory required to apply branching bisimulation algorithms are given where the time is taken to calculate the branching bisimulation equivalence classes starting with the τ-loop reduced transition systems. We compare the fastest known branching bisimulation algorithm headed "JGKW" with the algorithm presented in this paper. The row starting with 1394-fin-vvlarge reports on reducing the state space obtained from the mCRL2 model of the 1394 firewire protocol with 20 data elements as described in [3].

The results are obtained on a computer with an Intel Xeon Gold 6136 CPU at 3.00GHz processors and plenty of memory. Every benchmark was run 30 times. We removed the five fastest and the five slowest times and report the average of the remaining 20. Times are rounded to significant digits; trailing zeroes not followed by a decimal point are insignificant. The algorithm in this paper has almost always a better time and memory performance than JGKW. We also implemented optimisations, one of which performs initialisation without BLC sets. Never being detrimental, this further reduces the running time up to a factor 2. But as they are not in line with the presented algorithm we decided not to report them.

In both compared implementations states and label indices are stored as 64-bit numbers. We can substantially cut down on memory requirements by compressing these encodings, as is for instance done in [9], where incidentally an algorithm for strong bisimulation is compared to an algorithm for branching bisimulation applied to calculate a strong bisimulation reduction. As such comparisons are not very informative, we took care that the implementations of the two algorithms use the same implementation style. Both compared algorithms, along

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

with a number of others, can be tried in the freely available mCRL2 toolset [6] available at `https://www.mcrl2.org` and are part of the summer 2025 release. This toolset is open-source, which means that that the complex implementation code for these algorithms can be extracted and used elsewhere if so desired.

References

1 Rocco De Nicola and Frits Vaandrager. Action versus state based logics for transition systems. In I. Guessarian, editor, *Semantics of systems of concurrent processes: LITP spring school on theoretical computer science*, volume 469 of *LNCS*, pages 407–419. Springer, Berlin, 1990. doi:10.1007/3-540-53479-2_17.

2 Jean-Claude Fernandez. An implementation for an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2–3):219–236, 1990. doi:10.1016/0167-6423(90)90071-K.

3 Hubert Garavel and Bas Luttik. Four formal models of IEEE 1394 link layer. In Frédéric Lang and Matthias Volk, editors, *Proceedings Sixth Workshop on Models for Formal Analysis of Real System [MARS]*, volume 399 of *EPTCS*, pages 21–100, 2024. doi:10.4204/EPTCS.399.5.

4 Rob J. van Glabbeek and W. Peter Weijland. Branching time and abstraction in bisimulation semantics. *J. ACM*, 43(3):555–600, 1996. doi:10.1145/233551.233556.

5 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. doi:10.1145/3060140.

6 Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems*. MIT Press, 2014. URL: `https://mitpress.mit.edu/9780262547871/`.

7 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. doi:10.1007/BFb0032063.

8 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. doi:10.1007/978-3-662-49674-9_40.

9 Jules Jacobs and Thorsten Wißmann. Fast coalgebraic bisimilarity minimization. *Proc. ACM Program. Lang.*, 7(POPL):1514–1541, 2023. doi:10.1145/3571245.

10 David N. Jansen, Jan Friso Groote, Jeroen J.A. Keiren, and Anton Wijs. An $O(m \log n)$ algorithm for branching bisimilarity on labelled transition systems. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems: TACAS, Part II*, volume 12079 of *LNCS*, pages 3–20. Springer, Cham, 2020. doi:10.1007/978-3-030-45237-7_1.

11 Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, Berlin, 1980. doi:10.1007/3-540-10235-3.

12 Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987. doi:10.1137/0216062.

13 Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. doi:10.1137/0201010.

14 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. doi:10.1007/978-3-642-02424-5_9.