An O(mlog n) algorithm for computing stuttering equivalence and branching bisimulation

Citation for published version (APA):
Groote, J. F., Jansen, D. N., Keiren, J. J. A., & Wijs, A. J. (2017). An O(mlog n) algorithm for computing stuttering equivalence and branching bisimulation. *ACM Transactions on Computational Logic*, *18*(2), Article 13. https://doi.org/10.1145/3060140, https://doi.org/10.1145/3060140

Document license:
TAVERNE

DOI:
10.1145/3060140
10.1145/3060140

Document status and date:
Published: 01/06/2017

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

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

JAN FRISO GROOTE, Eindhoven University of Technology
DAVID N. JANSEN, Radboud Universiteit Nijmegen
JEROEN J. A. KEIREN, Open University of the Netherlands and Radboud Universiteit Nijmegen
ANTON J. WIJS, Eindhoven University of Technology

We provide a new algorithm to determine stuttering equivalence with time complexity $\mathcal{O}(m \log n)$, where n is the number of states and m is the number of transitions of a Kripke structure. This algorithm can also be used to determine branching bisimulation in $\mathcal{O}(m(\log |Act| + \log n))$ time, where Act is the set of actions in a labeled transition system.

Theoretically, our algorithm substantially improves upon existing algorithms, which all have time complexity of the form $\mathcal{O}(mn)$ at best. Moreover, it has better or equal space complexity. Practical results confirm these findings: they show that our algorithm can outperform existing algorithms by several orders of magnitude, especially when the Kripke structures are large.

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

CCS Concepts: • **Theory of computation** → *Formal languages and automata theory*; *Design and analysis of algorithms;*

Additional Key Words and Phrases: Branching bisimulation, algorithm

ACM Reference Format:
Jan Friso Groote, David N. Jansen, Jeroen J. A. Keiren, and Anton J. Wijs. 2017. An $\mathcal{O}(m \log n)$ algorithm for stuttering equivalence and branching bisimulation. ACM Trans. Comput. Logic 18, 2, Article 13 (June 2017), 34 pages.
DOI: http://dx.doi.org/10.1145/3060140

1. INTRODUCTION

Stuttering equivalence [Browne et al. 1988] and branching bisimulation [van Glabbeek and Weijland 1996] were proposed as alternatives to weak bisimulation [Milner 1980]. Weak and branching bisimulation are defined on labeled transition systems where some transitions are labeled with the hidden label τ. Stuttering equivalence is defined

Authors' addresses: J. F. Groote (corresponding author) and A. J. Wijs, Department of Mathematics and Computer Science, Eindhoven University of Technology, P. O. Box 513, 5600 MB Eindhoven, The Netherlands; email: {J.F.Groote, A.J.Wijs}@tue.nl; D. N. Jansen (Current address), Institute of Software, Chinese Academy of Sciences, Room 215, Building 5, Zhongguancun South Fourth Street #4, Beijing 100190, PR China; email: dnjansen@ios.ac.cn; J. J. A. Keiren, Open University of the Netherlands, Faculty of Management, Science & Technology, P. O. Box 2960, 6401 DL Heerlen, The Netherlands, and Radboud Universiteit, Institute for Computing and Information Sciences, P. O. Box 9010, 6500 GL Nijmegen, The Netherlands; email: Jeroen.Keiren@ou.nl.

DOI: http://dx.doi.org/10.1145/3060140

on Kripke structures where all transition labels are hidden, and the states are labeled. All these equivalences define whether two states exhibit the same visible behavior, where occurrences of hidden transitions cannot be observed.

The essential difference between branching bisimulation and stuttering equivalence on the one hand and weak bisimulation on the other hand is that in the former, when mimicking an a-transition, all states in a mimicking sequence $\tau^* a\, \tau^*$ must be related to the state either before or directly after the a from the first system. This means that branching bisimulation and stuttering equivalence are slightly stronger notions than weak bisimulation.

Groote and Vaandrager [1990] proposed an $\mathcal{O}(mn)$-time algorithm for stuttering equivalence and branching bisimulation, where n is the number of states and m is the number of transitions. We refer to this algorithm as GV. It is based on the $\mathcal{O}(mn)$ algorithm for strong bisimulation equivalence by Kanellakis and Smolka [1990]. Both algorithms require $\mathcal{O}(m)$ space. They calculate for each state whether it is (stuttering) bisimilar to another state.

The basic idea of both algorithms is to partition the set of states into blocks. The algorithms start from a (coarse) initial partition, which is iteratively refined. States that are stuttering equivalent/branching bisimilar invariantly reside in the same block. Whenever there are some states in a block B from which a transition is possible to some block SpB, which is called a splitter, and there are other states in B from which such a step is not possible, B is split accordingly, refining the partition. When no splitting is possible anymore, the partition is called stable, and two states are in the same block if and only if they are stuttering equivalent/branching bisimilar.

In order to determine whether a block B can be split, one needs to investigate whether there are sequences of transitions from states in block B to states in a splitter block SpB. A crucial observation by Groote and Vaandrager [1990] is that for stuttering equivalence and branching bisimulation, it is possible to determine this by only investigating single transitions from so-called bottom states. A bottom state is a state that has no outgoing transitions with a hidden label to a state in the same block. A prerequisite for the observation is that there are no cycles with hidden labels in a block. Fortunately, such cycles can be eliminated as a preprocessing step in linear time. It is noteworthy that for weak bisimulation, the notion of a bottom state does not help, making it far more expensive to determine whether blocks can be split.

Another important insight is the "process the smaller half" technique proposed by Hopcroft [1971] and Aho et al. [1974] and used by Paige and Tarjan [1987] and Fernandez [1990] for strong bisimulation. This allowed formulating partitioning algorithms with $\mathcal{O}(m \log n)$ worst-case time complexity. The essential idea is that a state with its incoming and outgoing transitions will only be revisited whenever it resides in a block of at most half the size of the current block. In particular, in the first visit, the state resides in a block with at most $n/2$ states. This implies that a state can only be visited $\lfloor \log_2 n \rfloor$ times.

In this article, we present the first algorithm for stuttering equivalence and branching bisimulation with $\mathcal{O}(m \log n)$ time complexity, in which all techniques mentioned earlier are combined. First, we select a splitter and establish which blocks it splits by combining the approach using bottom states from GV with the "process the smaller half" detection approach of Paige and Tarjan [1987]. Subsequently, we use the "process the smaller half" technique again to split each splittable block by only traversing transitions in a time proportional to the size of the smaller resulting subblock. As it is not known which of the two subblocks is smaller, the transitions of both subblocks are processed alternatingly, such that the total processing time can be attributed to the smaller block. For checking behavioral equivalences, applying such a technique is entirely new. We are only aware of one similar approach for an algorithm in which the smallest bottom strongly connected component of a graph needs to be found [Chatterjee and Henzinger 2011].

Groote and Vaandrager [1990] conjectured that the algorithm GV could be improved to time complexity $\mathcal{O}(mn_i + m_i n + m \log n)$, where n_i is the number of states that become bottom states and m_i is the number of hidden transitions, using the "process the smaller half" technique. Our results are much better than the improvement foreseen in this conjecture.

Although the basic sketch of the algorithm is relatively straightforward, meeting the complexity bound of $\mathcal{O}(m \log n)$ requires most operations in the algorithm to be $\mathcal{O}(1)$ time. In order to achieve this, the algorithm heavily relies on auxiliary data structures with subtle operations. We provide a detailed overview of the data structures and a pseudo-code description of the algorithm to facilitate implementation.

We implemented the algorithm and we used the implementation for two purposes. The first purpose was to increase our confidence that the algorithm is correct by running it on many thousands of randomly generated transition systems comparing the results with those of the classical GV algorithm. While carrying out these experiments, we checked a large number of invariants establishing that the integrity of the auxiliary data structures was maintained during all runs. We also carefully measured that the complexity of the algorithm was correct by assigning time budgets to each task in the algorithm and by monitoring that these were never depleted.

The second purpose of the implementation was to get an impression of the practical running times. From a theoretical viewpoint, our algorithm outperforms its predecessors substantially. But the theoretical bounds are complexity upper bounds, and depending on the transition systems, the classical algorithms could be much faster than the upper bound suggests. Furthermore, the increased bookkeeping to maintain the auxiliary data structures in the new algorithm may be such a burden that all gains are lost. We applied the algorithms on a substantial benchmark set and we found that our algorithm matches the best running times when transition systems are small. When the transition systems get large, our algorithm tends to outperform existing algorithms by several orders of magnitude.

The improved algorithm that we present here can also have a profound effect on the practical complexity to establish other behavioral equivalences. The existing algorithms for branching bisimulation/stuttering equivalence were already known to be practically very efficient. This is the reason that they are being used in multiple explicit-state model checkers, such as CADP [Garavel et al. 2012], the mCRL2 toolset [Groote and Mousavi 2014; Cranen et al. 2013], and TVT [Virtanen et al. 2004]. In particular, they are being used as preprocessing steps for other equivalences (weak bisimulation, trace-based equivalences) that are much harder to compute. By using the new algorithm, this preprocessing step can be sped up. Note that for weak bisimulation, an $\mathcal{O}(mn)$ algorithm was devised [Li 2009; Ranzato and Tapparo 2008], but until that time an expensive transitive closure operation of at best $\mathcal{O}(n^{2.373})$ was required. Additionally, the algorithms for deciding orthogonal bisimulation [Vu 2007] and the algorithm for governed stuttering bisimulation for parity games [Cranen et al. 2012] are based on GV, albeit more complicated. They might also be susceptible to the improvements presented in this article.

A preliminary version of this article appeared as Groote and Wijs [2016]. Compared to that version, the main changes in this article are found in Sections 5 and 6. In particular, Jansen and Keiren [2016] showed that the original algorithm did not meet the acclaimed $\mathcal{O}(m \log n)$ running time in certain degenerate cases. These problems have been resolved in this article. Furthermore, in this article, a detailed pseudocode description of the algorithm is presented, whereas Groote and Wijs [2016] gave a detailed, textual description. The data structures have been changed in order to obtain a cleaner representation while maintaining the required time complexities for all operations. The implementation reported on in Section 8 of this article is completely independent of the one reported on in Groote and Wijs [2016]. But the implementations performed comparably, with a major difference that the new one is on average 20% faster.

Related Work. The most relevant related work has already been mentioned [Groote and Vaandrager 1990; Kanellakis and Smolka 1990; Hopcroft 1971; Aho et al. 1974; Paige and Tarjan 1987]. There have been other attempts to improve upon GV. Blom and Orzan [2003] observed that GV only splits a block into two parts at a time. They proposed to split a block into as many parts as possible, reducing the burden of moving states and transitions to new blocks. They do this by computing signatures of states, capturing which blocks can be reached via outgoing transitions, and partitioning the states using a hash table based on those signatures. Furthermore, their algorithm does not explicitly replace blocks of the old partition with new blocks. Their worst-case time and space complexities are worse than those of GV; in particular, the algorithm runs in $\mathcal{O}(mn^2)$ time and $\mathcal{O}(mn)$ space, but in certain practical cases this algorithm performs much better than GV. Blom and van de Pol [2009] refined the ideas from Blom and Orzan [2003] to inductive signatures, improving the time complexity to $\mathcal{O}(n^2 + m)$ and the space complexity to $\mathcal{O}(m)$.

There is also work on improving the running times of these bisimulation reduction algorithms by employing multiple processors. For multicore CPUs, van Dijk and van de Pol [2016] provided an algorithm inspired by Blom and van de Pol [2009] for symbolic branching bisimulation minimization. Wijs [2015] proposed an algorithm for computing strong and branching bisimulation on Graphics Processing Units based on both GV [Groote and Vaandrager 1990] and the signature-based algorithms [Blom and Orzan 2003; Blom and van de Pol 2009]. These parallel and symbolic algorithms improve the required running time considerably in general, but they do not imply any improvement to the single-threaded algorithms.

Outline. Kripke structures and (divergence-blind) stuttering equivalence are introduced in Section 2. The rest of the article introduces our algorithm in increasing detail. First, in Section 3, we present a high-level description of the partition refinement approach due to Groote and Vaandrager [1990]. Next, the main ideas required to develop our $\mathcal{O}(m \log n)$ algorithm are introduced in Section 4. More details of how this complexity is achieved is described in Section 5. The algorithm requires most basic operations to be in $\mathcal{O}(1)$ time; we therefore introduce the necessary data structures and explain how the time bounds for these basic operations can be achieved in Section 6. We describe how our algorithm can be applied to compute branching bisimulation in Section 7. Finally, we evaluate the performance of our algorithm and compare it to existing algorithms in Section 8.

The reader who only wants to get a basic understanding of the key ideas in our algorithm can safely skip Sections 5 and 6. If the intent is to understand the details of the algorithm but not to implement it, Section 6 can safely be skipped.

2. PRELIMINARIES

We introduce Kripke structures and (divergence-blind) stuttering equivalence. Labeled transition systems and branching bisimulation will be addressed in Section 7. Note that in contrast to the standard definition [Kripke 1963], we do not require the transition relation of a Kripke structure to be total. On the one hand, our algorithm does not depend on the transition relation being total, and on the other hand, requiring it to be total would complicate our presentation in various places.

Definition 2.1 (*Kripke Structure*). A *Kripke structure* is a quadruple $(S, AP, \rightarrow, L)$, where

(1) S is a finite set of states,
(2) AP is a finite set of atomic propositions,
(3) $\rightarrow \subseteq S \times S$ is a transition relation, and
(4) $L : S \rightarrow 2^{AP}$ is a state labeling.

We use $n = |S|$ for the number of states and $m = |{\rightarrow}|$ for the number of transitions. We generally restrict our attention to Kripke structures where n is in $\mathcal{O}(m)$ as otherwise, the Kripke structure would necessarily contain uninteresting states without any incoming or outgoing transitions. The substructure containing the interesting states can be extracted in $\mathcal{O}(m)$ time. For sets of states $B, B' \subseteq S$, we write $s \rightarrow B$ if and only if there is some $s' \in B$ such that $s \rightarrow s'$, $B \rightarrow s$ if and only if there is some $s' \in B$ such that $s' \rightarrow s$, and $B \rightarrow B'$ if and only if there are states $s \in B, s' \in B'$ such that $s \rightarrow s'$. We write $s \not\rightarrow s'$, $s \not\rightarrow B$, and $B \not\rightarrow B'$ if and only if it is not the case that $s \rightarrow s'$, $s \rightarrow B$, and $B \rightarrow B'$, respectively. We generally use the abbreviations $in(s) = \{s' \mid s' \rightarrow s\}$ and $out(s) = \{s' \mid s \rightarrow s'\}$. Similarly, for a set of states $B \subseteq S$, we say that $in(B) = \{s \mid s \rightarrow B\}$ and $out(B) = \{s \mid B \rightarrow s\}$.

Definition 2.2 (*Divergence-blind Stuttering Equivalence*). Let $K = (S, AP, \rightarrow, L)$ be a Kripke structure. A symmetric relation $R \subseteq S \times S$ is a *divergence-blind stuttering equivalence* if and only if for all $s, t \in S$ such that $s\ R\ t$:

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

We say that two states $s, t \in S$ are *divergence-blind stuttering equivalent,* notation $s \leftrightarroweq_{dbs} t$, if and only if there is a divergence-blind stuttering equivalence relation R such that $s\ R\ t$.

An important property of divergence-blind stuttering equivalence is that divergent states (i.e., states with the same label on a loop) are divergence-blind stuttering equivalent. Furthermore, all such states are divergence-blind stuttering equivalent to a nondivergent state that can mimic all steps of the states on the loop. Stuttering equivalence is equal to divergence-blind stuttering equivalence, except that in addition, it distinguishes divergent and nondivergent states. We define stuttering equivalence in terms of divergence-blind stuttering equivalence by transforming the given Kripke structure.

Definition 2.3 (*Stuttering Equivalence*). Let $K = (S, AP, \rightarrow, L)$ be a Kripke structure. Define the Kripke structure $K_d = (S \cup \{s_d\}, AP \cup \{d\}, \rightarrow_d, L_d)$, where d is an atomic proposition not occurring in AP and s_d is a fresh state not occurring in S. Furthermore,

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

States $s, t \in S$ are *stuttering equivalent,* notation $s \leftrightarroweq_s t$, if and only if there is a divergence-blind stuttering equivalence relation R on S_d such that $s\ R\ t$.

Note that an algorithm for divergence-blind stuttering equivalence can also be used to determine stuttering equivalence by employing only a linear-time and -space transformation of the Kripke structure: According to Definition 2.3, we have to visit at most m transitions to find the self-loops and then add a single state s_d and at most $\min\{m, n\}$

transitions to s_d. Therefore, we only concentrate on an algorithm for divergence-blind stuttering equivalence.

3. PARTITIONS AND SPLITTERS: A SIMPLE ALGORITHM

Our algorithm to determine divergence-blind stuttering equivalence is based on partition refinement of partitions of the set of states S. Such a *partition* $\mathcal{P} = \{B_i \subseteq S \mid 1 \leq i \leq k\}$ is a set of nonempty subsets of states such that $B_i \cap B_j = \emptyset$ for all $1 \leq i < j \leq k$ and $S = \bigcup_{1 \leq i \leq k} B_i$. Each B_i is called a *block*.

If $s \rightarrow s'$ with states s and s' in the same block B, we call transition $s \rightarrow s'$ *inert w. r. t.* $\mathcal{P}$. We say that a partition $\mathcal{P}$ *coincides* with divergence-blind stuttering equivalence when $s \leftrightarroweq_{dbs} t$ if and only if there is a block $B \in \mathcal{P}$ such that $s, t \in B$. We say that partition $\mathcal{P}$ *respects* divergence-blind stuttering equivalence if and only if for all $s, t \in S$, if $s \leftrightarroweq_{dbs} t$, then there is some block $B \in \mathcal{P}$ such that $s, t \in B$. The goal of the algorithm is to calculate a partition that coincides with divergence-blind stuttering equivalence. This is done starting from an initial partition $\mathcal{P}_0$ consisting of blocks B satisfying $s, t \in B$ if and only if $L(s) = L(t)$. Constructing the initial partition takes time $\mathcal{O}(n)$. Note that this initial partition respects divergence-blind stuttering equivalence.

We say that a partition $\mathcal{P}$ is *cycle-free* if and only if there is no block $B \subseteq S$ with states $s_0, \ldots, s_k \in B$ for some $k \in \mathbb{N}$, such that $s_0 = s_k$ and $s_0 \rightarrow \cdots \rightarrow s_k$. It is easy to make the initial partition $\mathcal{P}_0$ cycle-free by merging all states on a cycle in each block into a single state and removing self-loops. This preserves divergence-blind stuttering equivalence and can be performed in linear time employing standard algorithms to find strongly connected components (see, e.g., Aho et al. [1974, Algorithm 5.4]).

We explain how partitions are refined. Given a block *RfnB* of the current partition and the union $\boldsymbol{SpC}$ of some of the blocks in the partition, we define

$$\begin{aligned} \mathit{split}(\mathit{RfnB}, \boldsymbol{SpC}) &= \{s \in \mathit{RfnB} \mid \exists k \in \mathbb{N}, s_0, \ldots, s_k \in S. \\ &\qquad s = s_0 \wedge (\forall i < k. s_i \rightarrow s_{i+1} \wedge s_i \in \mathit{RfnB}) \wedge s_k \in \boldsymbol{SpC}\} \\ \mathit{cosplit}(\mathit{RfnB}, \boldsymbol{SpC}) &= \mathit{RfnB} \setminus \mathit{split}(\mathit{RfnB}, \boldsymbol{SpC}) \end{aligned}$$

as the two blocks into which *RfnB* is split. We sometimes call $\mathit{split}(\mathit{RfnB}, \boldsymbol{SpC})$ the *red subblock* and $\mathit{cosplit}(\mathit{RfnB}, \boldsymbol{SpC})$ the *blue subblock* of *RfnB*. Note that if $\mathit{RfnB} \subseteq \boldsymbol{SpC}$, then *RfnB* is completely red since we can choose $k = 0$ in the definition. It is common to split with respect to single blocks, that is, $\boldsymbol{SpC}$ being a single block $\mathit{SpB} \in \mathcal{P}$ [Groote and Vaandrager 1990; Kanellakis and Smolka 1990], and we write $\mathit{split}(\mathit{RfnB}, \mathit{SpB})$ in that case. However, to obtain an $\mathcal{O}(m \log n)$ algorithm, Paige and Tarjan [1987] also split under the union of some of the blocks in $\mathcal{P}$.

We say that a block *RfnB* is *unstable under* $\boldsymbol{SpC}$ if and only if both $\mathit{split}(\mathit{RfnB}, \boldsymbol{SpC}) \neq \emptyset$ and $\mathit{cosplit}(\mathit{RfnB}, \boldsymbol{SpC}) \neq \emptyset$. In that case, we call $\boldsymbol{SpC}$ a *splitter* of *RfnB*. A partition $\mathcal{P}$ is *unstable under* $\boldsymbol{SpC}$ if and only if there is at least one $\mathit{RfnB} \in \mathcal{P}$ that is unstable under $\boldsymbol{SpC}$. If $\mathcal{P}$ is not unstable under $\boldsymbol{SpC}$, then it is called *stable under* $\boldsymbol{SpC}$. If $\mathcal{P}$ is stable under all $\boldsymbol{SpC}$, then it is simply called *stable*.

A *refinement* of $\mathit{RfnB} \in \mathcal{P}$ under $\boldsymbol{SpC}$ consists of the two new blocks $\mathit{split}(\mathit{RfnB}, \boldsymbol{SpC})$ and $\mathit{cosplit}(\mathit{RfnB}, \boldsymbol{SpC})$. A partition $\mathcal{P}'$ is a refinement of $\mathcal{P}$ under $\boldsymbol{SpC}$ if and only if all unstable blocks $\mathit{RfnB} \in \mathcal{P}$ have been replaced by new blocks $\mathit{split}(\mathit{RfnB}, \boldsymbol{SpC})$ and $\mathit{cosplit}(\mathit{RfnB}, \boldsymbol{SpC})$.

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

LEMMA 3.1. *Let $K = (S, AP, \rightarrow, L)$ be a Kripke structure and $\mathcal{P}$ a partition of S.*

(*1*) *For all states $s, t \in S$, if $s, t \in B$ with B a block of the partition $\mathcal{P}$, $\mathcal{P}$ is stable, and $\mathcal{P}$ is a refinement of the initial partition $\mathcal{P}_0$, then $s \underline{\leftrightarrow}_{dbs} t$.*

(*2*) *If $\mathcal{P}$ respects divergence-blind stuttering equivalence, then any refinement of $\mathcal{P}$ under the union of some of the blocks in $\mathcal{P}$ also respects it.*

(*3*) *If $\mathcal{P}$ is a cycle-free partition, then any refinement of $\mathcal{P}$ is also cycle-free.*

PROOF.

(1) We show that if $\mathcal{P}$ is a stable partition, the relation $R = \{\langle s, t\rangle \mid \exists B \in \mathcal{P}.s, t \in B\}$ is a divergence-blind stuttering equivalence. It is clear that R is symmetric. Assume $s\, R\, t$. Obviously, $L(s) = L(t)$ because $s, t \in B \in \mathcal{P}$ and $\mathcal{P}$ refines the initial partition. For the second requirement of divergence-blind stuttering equivalence, suppose $s \to s'$. There is a block B' such that $s' \in B'$. So $s \in split(B, B')$ by definition of *split*. As $\mathcal{P}$ is stable, also $t \in split(B, B')$, so there exist some $k \in \mathbb{N}$, $t_0, \ldots, t_{k-1} \in B$ and $t_k \in B'$ with $t = t_0 \to t_1 \to \cdots \to t_k$. This clearly shows that for all $i < k$, we have $s\, R\, t_i$ and $s'\, R\, t_k$. So, R is a divergence-blind stuttering equivalence, and therefore it holds for all states $s, t \in S$ that reside in the same block of $\mathcal{P}$ that $s \underline{\leftrightarrow}_{dbs} t$.

(2) The second part can be proven by reasoning toward a contradiction. Let us assume that a partition $\mathcal{P}'$ that is a refinement of $\mathcal{P}$ under an arbitrary union of blocks $\boldsymbol{SpC}$ does not respect divergence-blind stuttering equivalence, although $\mathcal{P}$ does. Hence, there are states $s, t \in S$ with $s \underline{\leftrightarrow}_{dbs} t$ and a block $RfnB \in \mathcal{P}$ with $s, t \in RfnB$ and s and t are in different blocks in $\mathcal{P}'$. Given that $\mathcal{P}'$ is a refinement of $\mathcal{P}$ under $\boldsymbol{SpC}$, w. l. o. g. $s \in split(RfnB, \boldsymbol{SpC})$ and $t \in cosplit(RfnB, \boldsymbol{SpC})$. By definition of *split*, there are $s_0, \ldots, s_{k-1} \in RfnB$ (for some $k \in \mathbb{N}$) and $s_k \in \boldsymbol{SpC}$ such that $s = s_0 \to s_1 \to \cdots \to s_k$. Then, either $k = 0$ and $RfnB \subseteq \boldsymbol{SpC}$, but then $t \notin cosplit(RfnB, \boldsymbol{SpC}) = \emptyset$, which is a contradiction, or $k > 0$, and since $s \underline{\leftrightarrow}_{dbs} t$, we can construct a sequence $t = t_0 \to \cdots \to t_1 \to \cdots \to t_k$ starting in t such that $s_i \underline{\leftrightarrow}_{dbs} t_i$ for all $i \leq k$. In particular, for $i < k$, there are $t_i = t_{i,0} \to \cdots \to t_{i,n_i} \in RfnB$ such that $s_i \underline{\leftrightarrow}_{dbs} t_{i,j}$ for $j \leq n_i$, and $t_{i,n_i} \to t_{i+1,0} = t_{i+1}$. For $s_{k-1} \to s_k$, similarly, a path $t_{k-1} = t_{k-1,0} \to \cdots \to t_{k-1,n_{k-1}} \to t_k$ can be constructed such that $t_{k-1,i} \underline{\leftrightarrow}_{dbs} s_{k-1}$ for all i, and $t_k \underline{\leftrightarrow}_{dbs} s_k$. Therefore, $t_k \in \boldsymbol{SpC}$, but this means that we have $t \in split(RfnB, \boldsymbol{SpC})$, again contradicting that $t \in cosplit(RfnB, \boldsymbol{SpC})$.

(3) If $\mathcal{P}$ is cycle-free, this property is straightforward, since splitting any block of $\mathcal{P}$ will only change inert transitions to noninert ones, and will thus not introduce cycles. □

This suggests Algorithm 1, which has time complexity $\mathcal{O}(mn)$ and space complexity $\mathcal{O}(m)$. It essentially was presented by Groote and Vaandrager [1990].

ALGORITHM 1: Basic partition refinement

```
$\mathcal{P} := \mathcal{P}_0$, i. e. the initial, cycle-free partition
while $\mathcal{P}$ is unstable under some block $SpB \in \mathcal{P}$ do
    $\mathcal{P} :=$ refinement of $\mathcal{P}$ under $SpB$
end while
return $\mathcal{P}$
```

By Lemma 3.1, it is an invariant of this algorithm that $\mathcal{P}$ respects divergence-blind stuttering equivalence and $\mathcal{P}$ is cycle-free. In particular, $\mathcal{P} = \mathcal{P}_0$ satisfies this invariant initially. If $\mathcal{P}$ is not stable, a refinement under some block SpB exists, splitting at least one block. Therefore, this algorithm finishes in at most $n - 1$ steps, as during each iteration of the algorithm the number of blocks increases by at least one, and the number of blocks can never exceed the number of states. When the algorithm terminates, $\mathcal{P}$ is stable, and therefore, two states are divergence-blind stuttering equivalent if and only if they are part of the same block in the final partition. This end result is independent of the order in which splitting took place.

In order to see that the time complexity of this algorithm is $\mathcal{O}(mn)$, we must show that we can both detect that $\mathcal{P}$ is unstable and carry out splitting in time $\mathcal{O}(m)$. The crucial observation to efficiently determine whether a partition is stable stems from Groote and Vaandrager [1990]. They showed that it is enough to look at the bottom states of a block, which always exist for each block because the partition is cycle-free. The *bottom states* of a block are those states that do not have an outgoing inert transition, that is, a transition to a state in the same block. They are defined by

$$bottom(B) = \{s \in B \mid s \not\to B\}.$$

The following lemma presents the crucial observation concerning bottom states.

LEMMA 3.2. *Let $K = (S, AP, \to, L)$ be a Kripke structure and $\mathcal{P}$ be a cycle-free partition of its states. Partition $\mathcal{P}$ is unstable under union **SpC** of some of the blocks in $\mathcal{P}$ if and only if there is a block $RfnB \in \mathcal{P}$ such that*

$$split(RfnB, \boldsymbol{SpC}) \neq \emptyset \text{ and } bottom(RfnB) \cap cosplit(RfnB, \boldsymbol{SpC}) \neq \emptyset.$$

PROOF.

$\Rightarrow$ If $\mathcal{P}$ is unstable under $\boldsymbol{SpC}$, then $split(RfnB, \boldsymbol{SpC}) \neq \emptyset$ and $cosplit(RfnB, \boldsymbol{SpC}) \neq \emptyset$. The first conjunct is identical to the first part of the right-hand side of the lemma. If $cosplit(RfnB, \boldsymbol{SpC}) \neq \emptyset$, there is a state $s \in cosplit(RfnB, \boldsymbol{SpC})$. As the blocks $RfnB \in \mathcal{P}$ do not have cycles, there is an $RfnB$-path from s to some bottom state $s_\perp \in RfnB$. We claim that $s_\perp \in cosplit(RfnB, \boldsymbol{SpC})$. Otherwise, $s_\perp$ would have a (strong) transition to $\boldsymbol{SpC}$, so $s_\perp \in split(RfnB, \boldsymbol{SpC})$ and therefore $s \in split(RfnB, \boldsymbol{SpC})$, which is a contradiction.

$\Leftarrow$ It follows from the right-hand side that $split(RfnB, \boldsymbol{SpC}) \neq \emptyset$ and $cosplit(RfnB, \boldsymbol{SpC}) \neq \emptyset$. □

This lemma can be used as follows to find a block to be split. Consider each $SpB \in \mathcal{P}$. Traverse its incoming transitions and mark the states that can reach SpB in zero or one step. If a block $RfnB$ has marked states, but not all of its bottom states are marked, the condition of the lemma applies, and it needs to be split. As shown by Groote and Vaandrager [1990], it is at most needed to traverse all transitions to carry this out, so its complexity is $\mathcal{O}(m)$.

If SpB is equal to $RfnB$, no splitting is possible. We implement it by marking all states in SpB, as each state in SpB can reach itself in zero steps. In this case, condition $bottom(RfnB) \cap cosplit(RfnB, \boldsymbol{SpC}) \neq \emptyset$ is not true. This is different from Groote and Vaandrager [1990], where a block is never considered as a splitter of itself. Our approach is more convenient in the algorithm in the next sections.

If a block $RfnB$ is unstable and all states from which a state in SpB can be reached in one step are marked, then a straightforward recursive procedure is required to extend the marking to all states in $split(RfnB, SpB)$, and those states need to be moved to a new block. This takes time proportional to the number of incoming transitions of $RfnB$; that is, extending the marking in all unstable blocks and if needed moving the states to a new block take at most $\mathcal{O}(m)$ time.

4. OUTLINE OF THE $\mathcal{O}(m \log n)$ ALGORITHM

The crucial idea to transform the algorithm from the previous section into an $\mathcal{O}(m \log n)$ algorithm stems from Paige and Tarjan [1987]. In addition to the current partition $\mathcal{P}$, a coarser partition $\mathcal{C}$ is maintained. We call the elements of $\mathcal{C}$ *constellations*. A constellation is the union of a set of blocks in $\mathcal{P}$. If it corresponds with a single block, the constellation is called *trivial*. The algorithm maintains the following invariant:

INVARIANT 4.1. *The current partition $\mathcal{P}$ is stable under each constellation in $\mathcal{C}$.*

The initial set of constellations $\mathcal{C}_0$ contains a single constellation with all states, which satisfies the invariant. The desire is to make every constellation trivial, as this implies that the partition $\mathcal{P}$ is stable, allowing the algorithm to finish. So, assume that some constellation $\boldsymbol{SpC}$ is not trivial, which means it consists of two or more blocks, among which $SpB \in \mathcal{P}$. We want to split $\boldsymbol{SpC}$ into SpB and $\boldsymbol{SpC} \setminus SpB$. We then have to refine $\mathcal{P}$ under both SpB and $\boldsymbol{SpC} \setminus SpB$ to re-establish the invariant.

According to the "process the smaller half" principle, it is only allowed to spend time proportional to the smaller splitter (SpB or $\boldsymbol{SpC} \setminus SpB$) and its transitions to determine whether the current partition can be refined under both splitters. We always select SpB such that $|SpB| \leq \frac{1}{2}|\boldsymbol{SpC}|$. Determining the blocks that must be split by SpB can be performed in exactly the same way as in the $\mathcal{O}(mn)$ algorithm.

We now turn our attention to identifying the blocks that must be split under $\boldsymbol{SpC} \setminus SpB$. By the invariant, only blocks of the shape $split(RfnB, SpB)$ can be unstable under $\boldsymbol{SpC} \setminus SpB$. When investigating a transition from $RfnB$ to SpB, it is stored whether there are also transitions from its source state in $RfnB$ to $\boldsymbol{SpC} \setminus SpB$. This is possible as the data structures allow quickly finding all transitions from a state to a constellation. When all such transitions into SpB have been investigated, it is easy to find out whether there are bottom states in $split(RfnB, SpB)$ from which $\boldsymbol{SpC} \setminus SpB$ cannot be reached (Lemma 3.2, right-hand side). Efficiently checking whether there are other (possibly nonbottom) states in $split(RfnB, SpB)$ with a transition to $\boldsymbol{SpC} \setminus SpB$ (Lemma 3.2, left-hand side) is not trivial and requires additional data structures. The main observation is that in order to check that the current partition is stable under $\boldsymbol{SpC} \setminus SpB$, we only need to check the stability of $split(RfnB, SpB)$ under $\boldsymbol{SpC} \setminus SpB$, and this can be done in time linear in the size of SpB.

The part of the algorithm explained previously is provided in detail as Algorithm 2, which we discuss completely in Section 5. As observed before, the order in which splitting is performed does not affect the correctness of its outcome. Based on Groote and Vaandrager [1990], we therefore immediately get the following result.

THEOREM 4.2. *The algorithm is correct; that is, the final partition $\mathcal{P}$ coincides with divergence-blind stuttering equivalence.*

From the previous description, it should be clear that every individual state is part of some splitter SpB at most $\lfloor \log_2 n \rfloor$ times, so refining $\mathcal{C}$ and detecting whether blocks in the current partition can be split fit in the $\mathcal{O}(m \log n)$ bound. Actually carrying out the splitting with the same time complexity requires solving two more issues.

(1) To refine a block $RfnB$ in the setting of strong bisimulation, it is enough to move the smaller subblock, either the marked states (the red states) or the unmarked states (the blue states), to a new block and leave the other states in the old block. In the setting of stuttering equivalence, not only the marked states are red but also the states that can reach marked states through inert transitions in the block $RfnB$. Therefore, the marking has to be extended. This extension must be carried out in time proportional to the size of the smaller subblock of $RfnB$ to comply with the "process the smaller half" principle. A priori it is not known whether the red or the blue subblock is the smaller one. Even worse, when the red subblock is larger, there is not even time available to mark all red states.

We solve this problem using a remarkable technique consisting of two coroutines running alternatingly to identify the smaller block resulting from the refinement. For

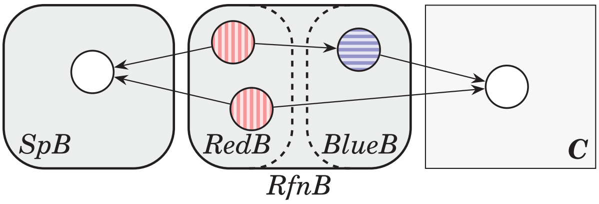


Fig. 1. After splitting *RfnB* under *SpB*, *RedB* is not stable under ***C***.

the red states, the marking is extended with a simple recursive algorithm. For the blue states, it is determined that they cannot be marked by establishing that all the outgoing inert transitions go to blue states. The whole operation stops when either all red or all blue states have been found and therefore runs in time proportional to the number of transitions of the smaller block. The new block is created from this smaller block.

This procedure ensures that whenever a state is split off from a block *RfnB*, it becomes part of a block that is at most half the size of *RfnB*. Therefore, each state can be split off at most $\lfloor \log_2 n \rfloor$ times. The details are in Algorithm 3, which is explained in detail in the next section.

(2) If blocks are split, the new partition is not automatically stable under all constellations. This is contrary to the situation in Paige and Tarjan [1987] and was already observed by Groote and Vaandrager [1990]. Figure 1 illustrates this situation. Block *RfnB* is stable under constellation $\boldsymbol{C}$. But if *RfnB* is split under block *SpB* into *RedB* and *BlueB*, block *RedB* is not stable under $\boldsymbol{C}$. In general, a refined partition may not be stable under all constellations if and only if the refinement introduced new bottom states. In Figure 1, the top state in *RedB* was a nonbottom state in *RfnB* but became a bottom state in $\mathit{RedB} = \mathit{split}(\mathit{RfnB}, \mathit{SpB})$. Blocks in which no new bottom states are introduced remain stable, as shown by the following lemma.

LEMMA 4.3. *Let $K = (S, AP, \rightarrow, L)$ be a Kripke structure with cycle-free partition $\mathcal{P}$ and refinement $\mathcal{P}'$. If $\mathcal{P}$ is stable under a constellation $\boldsymbol{C}$ and RfnB $\in \mathcal{P}$ is refined into $\mathit{RfnB}_1, \ldots, \mathit{RfnB}_k \in \mathcal{P}'$, then for each RfnB_i whose bottom states are also bottom states in RfnB, it holds that RfnB_i is also stable under $\boldsymbol{C}$.*

PROOF. Assume RfnB_i is not stable under $\boldsymbol{C}$. This means that $\mathit{split}(\mathit{RfnB}_i, \boldsymbol{C}) \neq \emptyset$ and $\mathit{bottom}(\mathit{RfnB}_i) \cap \mathit{cosplit}(\mathit{RfnB}_i, \boldsymbol{C}) \neq \emptyset$. Hence, there is a state $s \in \mathit{RfnB}_i$ such that $s \rightarrow \boldsymbol{C}$ and there is a bottom state $t \in \mathit{RfnB}_i$ with $t \not\rightarrow \boldsymbol{C}$. As *RfnB* is stable under $\boldsymbol{C}$ and $s \rightarrow \boldsymbol{C}$, all bottom states in *RfnB* must have at least one transition to a state in $\boldsymbol{C}$. Therefore, t cannot be a bottom state of *RfnB* and must have become a bottom state after splitting *RfnB*. □

This means that if a block $\hat{B}$ is the result of a refinement and some of its states have become new bottom states, it must be checked whether $\hat{B}$ is stable under the constellations. Typically, from the new bottom states, a subset of the constellations reachable from the original block can be reached. We first separate, in a similar refinement step as described previously, the new bottom states from the old ones. Then we have to refine the subblock containing the new bottom states under all constellations it can reach. To do these refinements in an orderly manner, the reachable constellations are inserted into and removed from a search tree. The complexity of determining how $\hat{B}$ must be split is $\mathcal{O}(\log n)$ times the number of outgoing transitions of the new bottom states. The details are in Algorithm 4 and are discussed in Section 5.

The time complexity calculation of our algorithm is based on the following theorem, which we prove along with explaining the details of the algorithm in the next section.

THEOREM 4.4. *Every step in the algorithm falls under one of the following three cases:*

(*1*) *Every state s is moved to a new constellation (while identifying which blocks are unstable under this new constellation) at most $\lfloor \log_2 n \rfloor$ times. Whenever this happens, $\mathcal{O}(|in(s)| + |out(s)|)$ time is spent on this state.*
(*2*) *Similarly, every state s is moved to a new block during a refinement of a block at most $\lfloor \log_2 n \rfloor$ times. Whenever this happens, $\mathcal{O}(|in(s)| + |out(s)|)$ time is spent on this state.*
(*3*) *Every state s becomes a new bottom state at most once. Whenever this happens, $\mathcal{O}((|in(s)| + |out(s)|) \log n)$ time is spent on this state.*

Aggregating these complexities over all states implies that the total time complexity is $\mathcal{O}(m \log n)$.

ALGORITHM 2: Main loop of partition refinement for divergence-blind stuttering equivalence

```
function dbStutteringEquivalence($S, AP, \rightarrow, L$)
{Find the divergence-blind stuttering equivalence classes for Kripke structure ($S, AP, \rightarrow, L$)
with $n \in \mathcal{O}(m)$.}
$\mathcal{P} := \mathcal{P}_0$, i. e. the initial, cycle-free partition; $\mathcal{C} := \{S\}$          } $\mathcal{O}(m \log n)$
Initialise all temporary data                                                                        }
while $\mathcal{C}$ contains a non-trivial constellation $\boldsymbol{SpC}$ do                       } $\leq n$ iterations
    Choose a small splitter block $SpB \subset \boldsymbol{SpC}$ from $\mathcal{P}$, i. e. $|SpB| \leq \frac{1}{2}|\boldsymbol{SpC}|$   }
    Create a new constellation $\boldsymbol{NewC}$ and move $SpB$ from $\boldsymbol{SpC}$ to $\boldsymbol{NewC}$   }
    $\mathcal{C}$ := partition $\mathcal{C}$ where $SpB$ is removed from $\boldsymbol{SpC}$ and $\boldsymbol{NewC}$ is added   } $\mathcal{O}(1)$
    Mark block $SpB$ as refinable                                                                     }
    Mark all states of $SpB$ as predecessors                                                          }
    for all $s \in SpB$ do {Find predecessors of $SpB$}                                               }
        for all $s' \in in(s) \setminus SpB$ do                                                       }
            Mark the block of $s'$ as refinable                                                       }
            Mark $s'$ as predecessor of $SpB$                                                         }
            Register that $s' \rightarrow s$ goes to $\boldsymbol{NewC}$ (instead of $\boldsymbol{SpC}$)   } $\mathcal{O}\left(\begin{matrix}|in(SpB)|+\\|out(SpB)|\end{matrix}\right)$
            Store whether $s'$ still has some transition to $\boldsymbol{SpC} \setminus SpB$          }
        end for                                                                                       }
        Register that inert transitions from $s$ go to $\boldsymbol{NewC}$ (instead of $\boldsymbol{SpC}$)   }
        Store whether $s$ still has some transition to $\boldsymbol{SpC} \setminus SpB$               }
    end for                                                                                           }
    for all refinable blocks $RfnB$ do {Stabilise the partition again}                               } $\leq |in(SpB)|$ iterations
        Mark block $RfnB$ as non-refinable                                                            } $\mathcal{O}(1)$
        $\langle RedB, BlueB \rangle$ := Refine($RfnB, \boldsymbol{NewC}$, {marked states $\in RfnB$}, $\emptyset$)
        if $RedB$ contains new bottom states then
            $RedB$ := PostprocessNewBottom($RedB, BlueB$)
        end if
        $\langle RedB, BlueB \rangle$ := Refine($RedB, \boldsymbol{SpC} \setminus SpB, \emptyset$, {transitions $RedB \rightarrow$
                                              $\boldsymbol{SpC} \setminus SpB$})
        if $RedB$ contains new bottom states then
            PostprocessNewBottom($RedB, BlueB$)
        end if
        Unmark all states of the original $RfnB$ as predecessors                                      } $\mathcal{O}(1)$
    end for
end while
return $\mathcal{P}$
```

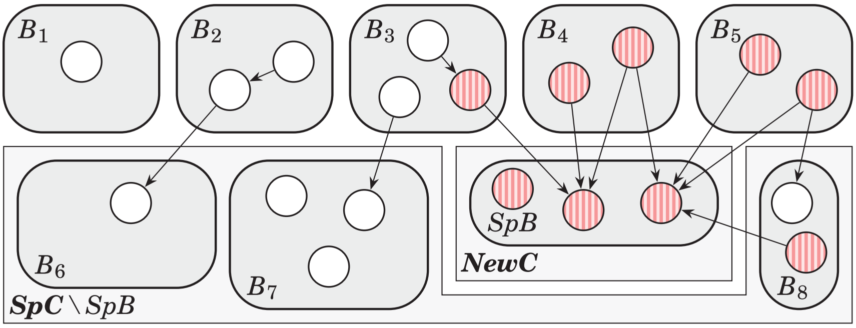


Fig. 2. dbStutteringEquivalence marks predecessor states and blocks.

5. DETAILED EXPOSITION OF THE $\mathcal{O}(m \log n)$ ALGORITHM

In the previous section, we have given a high-level description of the algorithm, and we have illustrated the main ideas to obtain the $\mathcal{O}(m \log n)$ bound on time complexity. In the current section, we explain the details of our algorithm using the pseudocode in Algorithms 2, 3, and 4.

The pseudocode is annotated with time budgets. For time complexities, we write $|in(B)| = \sum_{s \in B} \max\{1, |in(s)|\}$ and $|out(B)| = \sum_{s \in B} \max\{1, |out(s)|\}$. To meet the desired complexity bound, most basic operations need to run in $\mathcal{O}(1)$ time; when a basic operation is slower, we mention its complexity separately. Treatment of the data structures used to achieve this complexity is deferred to Section 6. We use the abbreviations $in_\tau(s)$ and $out_\tau(s)$ to denote the incoming and outgoing transitions that are inert according to the current partition $\mathcal{P}$, respectively.

5.1. Refining Constellations

After initialization, Algorithm 2 refines the partition $\mathcal{P}$ in lines 2.4 to 2.32 until it coincides with $\mathcal{C}$ and therefore has become stable. If the partition is not stable, there is at least one nontrivial constellation $\boldsymbol{SpC}$ (a constellation that contains more than one block). We select one block $SpB \subset \boldsymbol{SpC}$ that is at most half the size of $\boldsymbol{SpC}$ in line 2.5. We first move SpB from $\boldsymbol{SpC}$ to a new trivial constellation $\boldsymbol{NewC}$ and reduce the constellation $\boldsymbol{SpC}$ to contain the states in $\boldsymbol{SpC} \setminus SpB$.

Since we have split $\boldsymbol{SpC}$, Invariant 4.1 may be violated now. To re-establish the invariant, we refine the current partition under SpB and $\boldsymbol{SpC} \setminus SpB$. First, we mark all states that can reach SpB in zero or one step by traversing all incoming, noninert transitions of SpB (lines 2.10 to 2.19). Each block that contains a marked state possibly needs to be split; we call these refinable blocks. We visit all such blocks $RfnB$ in lines 2.20 to 2.31. If we look closely at these blocks, they contain marked and unmarked, bottom and nonbottom states, constituting four subsets. Each block can possibly be split into multiple subblocks.

Example 5.1. In Figure 2, a typical situation is depicted. The constellation $\boldsymbol{SpC}$ consists of four blocks B_6, B_7, SpB, and B_8. The block SpB is moved out of $\boldsymbol{SpC}$ to its own constellation $\boldsymbol{NewC}$. The states that can reach SpB in zero or one transition are marked in red ◍. Note that blocks B_1 and B_2 are not touched, and these can be ignored as their stability is untouched when refining constellation $\boldsymbol{SpC}$: block B_1 has no transition to $\boldsymbol{SpC}$ at all, and all transitions from B_2 to $\boldsymbol{SpC}$ go to $\boldsymbol{SpC} \setminus SpB$. Blocks B_3 and B_8 must be split as they contain some states that can reach SpB and some states that cannot. The blocks B_4, B_5, and also SpB are stable under SpB, but perhaps they still need to be split as they may be unstable under $\boldsymbol{SpC} \setminus SpB$. It is described next how this is done. Concretely, B_4 and SpB are stable, but B_5 needs to be split further.

Refining a block $RfnB$ with marked states under splitter $\boldsymbol{NewC} = SpB$ is done in Refine in line 2.22. We rely on the markings of the states here: the marked states are known to be in $split(RfnB, SpB)$, and the unmarked bottom states are known to be in $cosplit(RfnB, SpB)$. For the remaining states, Refine determines to which subset they belong and splits $RfnB$ accordingly. Intuitively, the states that become red in Refine are the states that can reach SpB, and the blue states are those that *cannot* reach it. The time taken by Refine is proportional to the number of transitions of the smaller subset, and Refine returns $split(RfnB, SpB)$ and $cosplit(RfnB, SpB)$. We elaborate on the details of Refine in Section 5.2.

As noted by Paige and Tarjan [1987], the blue subblock $cosplit(RfnB, SpB)$ after the first call to Refine is stable under $\boldsymbol{SpC} \setminus SpB$. Therefore, only $RedB = split(RfnB, SpB)$ needs to be refined further under $\boldsymbol{SpC} \setminus SpB$. Even if $RfnB$ is stable under SpB, it is not necessarily stable under $\boldsymbol{SpC} \setminus SpB$. In line 2.26, we refine $RedB$ under $\boldsymbol{SpC} \setminus SpB$. In this block, all states can reach SpB, and we need to identify those states that can also reach $\boldsymbol{SpC} \setminus SpB$. Here, the marking does not provide enough information to start the refinement as we need to identify the bottom states in $RedB$ that have a transition to $\boldsymbol{SpC} \setminus SpB$ to determine the initially known red and blue states. Therefore, we take all transitions from $RedB$ to $\boldsymbol{SpC} \setminus SpB$ as a starting point. The amount of work we need to do is limited by the number of transitions of the smaller resulting block, as we illustrate next.

We maintain a special data structure to recall for any block B and constellation $\boldsymbol{C}$ which transitions go from B to $\boldsymbol{C}$. While visiting transitions from $RfnB$ to SpB to mark states, we adapt this data structure to distinguish the transitions to $\boldsymbol{NewC} = SpB$ from those to $\boldsymbol{SpC} \setminus SpB$ in lines 2.14 and 2.17. Note that in line 2.17, the program may spend $\mathcal{O}(|out(s)|)$ time. We use this data structure to traverse the transitions from $RedB$ to $\boldsymbol{SpC} \setminus SpB$; the sources of these transitions are known to be in $split(RedB, \boldsymbol{SpC} \setminus SpB)$, and they are the states that are initially red in the second call to Refine. On the other hand, we also traverse all bottom states of $RedB$ and check whether they have outgoing transitions to $\boldsymbol{SpC} \setminus SpB$. For marked states, in particular all bottom states in $RedB$, this information is stored in lines 2.15 and 2.18, so this takes $\mathcal{O}(1)$ time per bottom state in $RedB$. This provides us with the bottom states known to be in $cosplit(RedB, \boldsymbol{SpC} \setminus SpB)$, the initially blue states in the second call to Refine. Note that all bottom states in $RedB$ have been marked before the first call to Refine, so we can traverse them once more without increasing the time complexity. Refine will traverse these states and transitions as earlier to compute the initially red and blue states, then decide whether the remaining states of $RedB$ are red or blue (w. r. t. $\boldsymbol{SpC} \setminus SpB$) and refine the block as before.

As illustrated by Lemma 4.3, blocks that have not gained any new bottom states are stable under the new partition. However, the remaining blocks—the ones that did gain some new bottom states—may not yet be stable; we therefore need to apply a postprocessing step to re-establish the invariant for these blocks. This is done in lines 2.24 and 2.28. It boils down to again refining these blocks, and is described in more detail in Section 5.3.

5.2. Refining Blocks

A description of procedure Refine($RfnB, SpB, \ldots$) is given in Algorithm 3, where the two columns present two separate coroutines. In the following discussion, we refer with $3.n\ell$ to the operation in the left column of line n in Algorithm 3, and with $3.nr$ to the operation in the right column of line n. Using the two coroutines, the algorithm simultaneously

ALGORITHM 3: Refine a block under ***SpC***

```
3.1  function REFINE($RfnB$, $\boldsymbol{SpC}$, $Red$, $FromRed$)
     {Try to refine block $RfnB$, depending on whether states have (weak) transitions to the
     splitter constellation $\boldsymbol{SpC}$. States in $Red$ are known to have such a transition; alternatively,
     $FromRed$ contains all strong transitions from $RfnB$ to $\boldsymbol{SpC}$. If $FromRed \neq \emptyset$, then bottom
     states that are not in $Red$ can be tested quickly whether they have such a transition.}
3.2  if $RfnB \subseteq \boldsymbol{SpC}$ then return $\langle RfnB, \emptyset \rangle$                          }
3.3  $Test$ := {bottom states} $\setminus Red$,   $Blue := \emptyset$                            } $\mathcal{O}(1)$
3.4  begin {Spend the same amount of work on either coroutine:}                   }
3.5    whenever $|Blue| > \frac{1}{2}|RfnB|$ then       ||  whenever $|Red| > \frac{1}{2}|RfnB|$ then      } $\mathcal{O}(1)$ per assignment
                         Abort this coroutine      ||                     Abort this coroutine      } to $Blue$ or $Red$, resp.
3.6    while $Test \neq \emptyset \wedge FromRed \neq \emptyset$ do  ||  while $FromRed \neq \emptyset$ do            }
3.7      Choose $s \in Test$                       ||    Choose $s \to t \in FromRed$              }
3.8      if $s \to \boldsymbol{SpC}$ then                    ||    $Test := Test \setminus \{s\}$               }
3.9        Move $s$ from $Test$ to $Red$              ||    $Red := Red \cup \{s\}$                    } $\mathcal{O}(|Test|)$
3.10     else                                      ||    $FromRed := FromRed \setminus \{s \to t\}$   } and
3.11       Move $s$ from $Test$ to $Blue$             ||                                          } $\mathcal{O}(|FromRed|)$
3.12     end if                                    ||                                          }
3.13   end while                                   ||  end while                                 }
3.14   $Blue := Blue \cup Test$                        ||                                          }
3.15   while $Blue$ contains                         ||  while $Red$ contains                        }
                           unvisited states do     ||                      unvisited states do  }
3.16     Choose an unvisited $s \in Blue$             ||    Choose an unvisited $s \in Red$            }
3.17     Mark $s$ as visited                         ||    Mark $s$ as visited                        }
3.18     for all $s' \in in_\tau(s) \setminus Red$ do          ||    for all $s' \in in_\tau(s)$ do                 } $\mathcal{O}\left(\begin{array}{l}|in(NewB)| + \\ |out(NewB)| + \\ |out(NewBott)|\end{array}\right)$
3.19       if $notblue(s')$ undefined then           ||                                          }
3.20         $notblue(s') := |out_\tau(s')|$              ||                                          }
3.21       end if                                  ||                                          } and
3.22       $notblue(s') := notblue(s') - 1$             ||                                          } $\mathcal{O}(|in(NewB)|)$
3.23       if $notblue(s') = 0 \wedge (FromRed =$        ||                                          }
                    $\emptyset \vee s' \not\to \boldsymbol{SpC})$ then    ||                                          }
3.24         $Blue := Blue \cup \{s'\}$                  ||      $Red := Red \cup \{s'\}$                  }
3.25       end if                                  ||                                          }
3.26     end for                                   ||    end for                                 }
3.27   end while                                   ||  end while                                 }
3.28   Abort the other coroutine                   ||  Abort the other coroutine                 }
3.29   Move $Blue$ to a new block $NewB$              ||  Move $Red$ to a new block $NewB$             } $\mathcal{O}(|out(NewB)|)$
3.30   Destroy all temporary data                  ||  Destroy all temporary data                } as lines 3.6 to 3.27
3.31   for all $s \in NewB$ do                        ||  for all non-bottom $s \in NewB$ do            }
3.32     for all $s' \in in_\tau(s) \setminus NewB$ do        ||    for all $s' \in out_\tau(s) \setminus NewB$ do    }
3.33       $s' \to s$ is no longer inert               ||      $s \to s'$ is no longer inert             } $\mathcal{O}(|in(NewB)|)$
3.34       if $|out_\tau(s')| = 0$ then                 ||    end for                                 } or
3.35         $s'$ is a new bottom state               ||    if $|out_\tau(s)| = 0$ then                  } $\mathcal{O}(|out(NewB)|)$
3.36       end if                                  ||      $s$ is a new bottom state               }
3.37     end for                                   ||    end if                                  }
3.38   end for                                     ||  end for                                   }
3.39   $RedB := RfnB$,   $BlueB := NewB$             ||  $RedB := NewB$,   $BlueB := RfnB$           } $\mathcal{O}(1)$
3.40 end                                                                                 }
3.41 $\mathcal{P}$ := partition $\mathcal{P}$ where $NewB$ is added and the states in $NewB$ are removed from $RfnB$
3.42 return $\langle RedB, BlueB \rangle$ (with old and new bottom states separated)
```

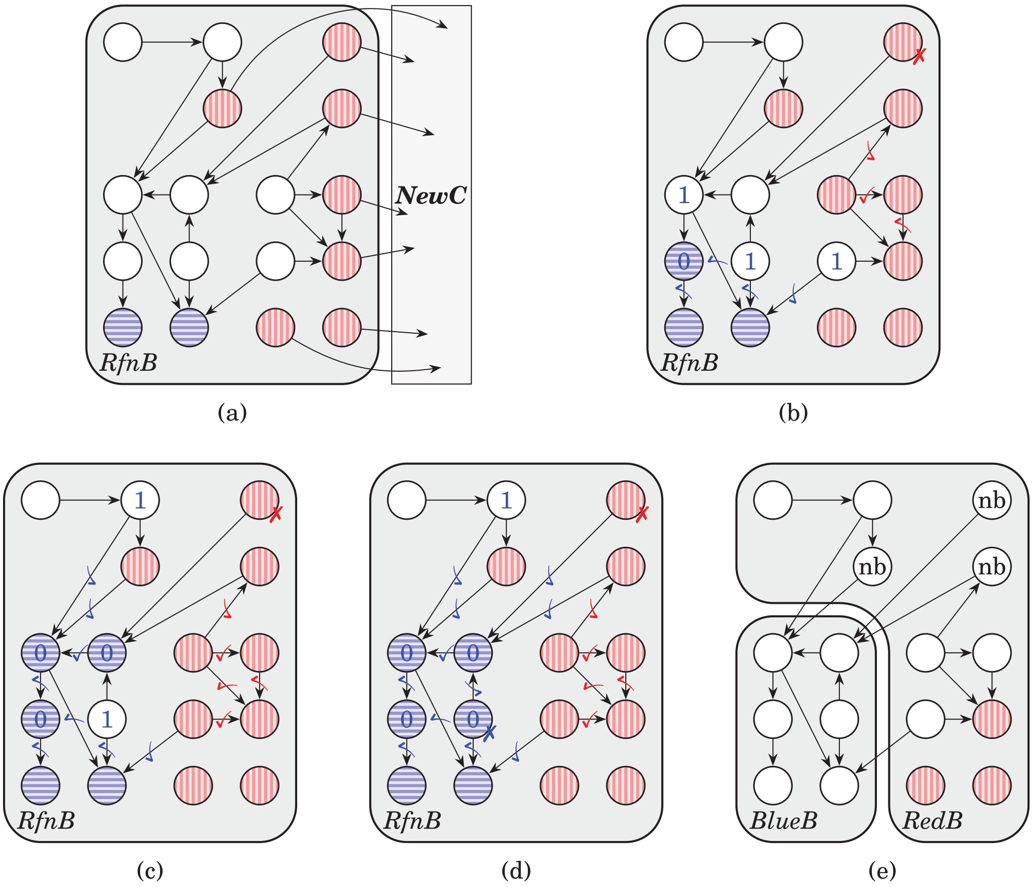
colors states in *RfnB* red or blue. Red means that there is a path from the state in *RfnB* to a state in *SpB*. Red states will end up in *split*(*RfnB*, *SpB*). Blue means that there is no such path and hence blue states will end up in *cosplit*(*RfnB*, *SpB*). For the moment, ignore lines 3.6 to 3.13. These lines are skipped when REFINE is called from line 2.22, since there, REFINE is called with $FromRed = \emptyset$. Initially, when called from line 2.22, marked states are red and unmarked bottom states are blue. This coloring is extended to all states in $RfnB$, spending equal time on each color. The procedure is stopped as soon as all states of one of the colors have been visited; that is, that color can no longer be extended. We color states red whenever they are a predecessor of a red state; this requires time proportional to the number of inert incoming transitions of red states. States are colored blue whenever it can be determined that all outgoing inert transitions go to blue states and there is no direct transition to SpB. To do this efficiently, for each state s', we keep track of the number of inert transitions to nonblue states that s' has. Whenever we visit blue states, we decrement $notblue(s')$ for each predecessor s' in $RfnB$. A state s' can be colored blue as soon as $notblue(s') = 0$. Since we only visit predecessors of blue states along inert transitions, and for each of these states we do $\mathcal{O}(1)$ work, this requires time proportional to the number of inert incoming transitions of blue states.

The algorithm balances the work by alternating between the two coroutines. Note that one should take care where to interrupt a coroutine; in particular, it should not be interrupted between lines 3.18ℓ and 3.23ℓ, because if during the execution of those operations state s' is being added to Red and $FromRed$ is emptied in lines 3.9r to 3.10r, s' might erroneously also be colored blue. On the other hand, it is always safe to switch at the end of a loop iteration before checking the loop condition.

The coroutine that finishes coloring first, provided that its number of colored states does not exceed $\frac{1}{2}|RfnB|$, has completely colored the smaller block resulting from the refinement; the other coroutine can be aborted. Since the work done so far has been balanced between the two coroutines, all work can be attributed to the smaller block, so the time used for the refinement so far is proportional to the number of incoming transitions of the smaller resulting block. We move the states of this smaller block to a newly created block.

Before returning, Refine also may find new bottom states. These are states of which all previously inert transitions now go from red to blue states (there cannot be transitions from blue to red states). Depending on which subblock is smaller, these transitions are visited forward or backward in lines 3.31 to 3.38, so the time used is proportional to the transitions of the smaller subblock. When a new bottom state is found, lines 3.35ℓ and 3.36r move it from the nonbottom states to a separate part of the bottom states of its block.

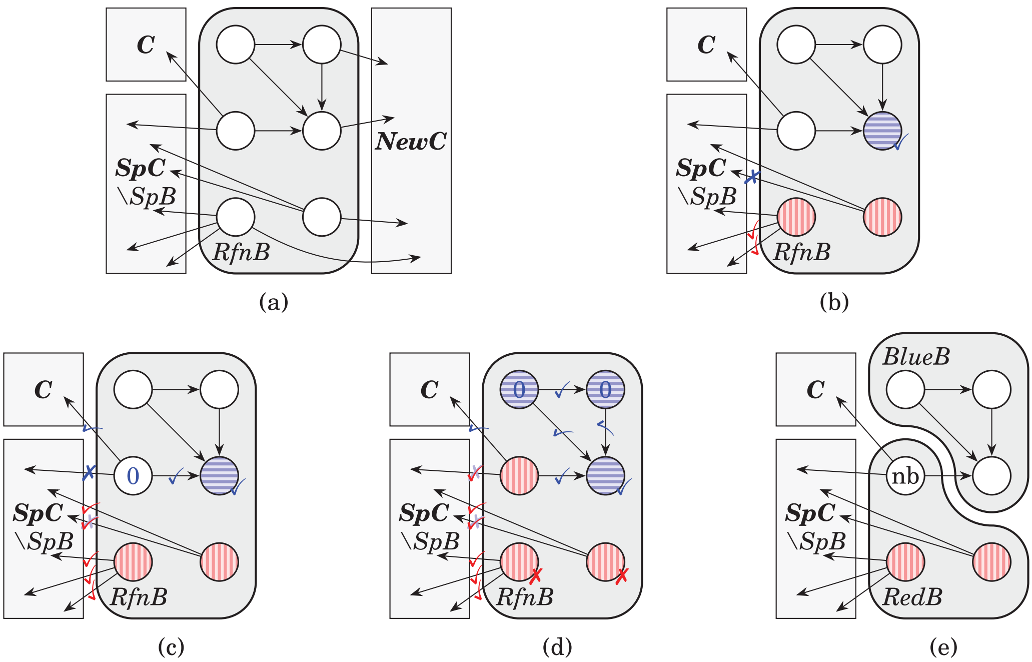
Example 5.2. The previous operation of Refine is illustrated in Figure 3. Figure 3(a) shows the situation when calling Refine from line 2.22: states with a transition to ***NewC*** are marked red ⦶, and other bottom states are blue ⊜. In Figure 3(b), every coroutine has considered four transitions. (In a single-thread implementation, the sequence might be 1 × blue, 2 × red, 2 × blue, 2 × red, 1 × blue). The blue coroutine found three states, indicated with "1," with one inert transition each that do not (yet) point to blue states, and one state, indicated with "0," that has no such transition. The numbers correspond with the *notblue*-value of the respective state. When a *notblue*-value becomes 0, the state changes its color to blue ⊜. The red coroutine has visited one state without incoming transitions (indicated with "***x***"); for this state, no time is spent in lines 3.18r to 3.26r, but still some time is spent in the outer loop (lines 3.15r to 3.27r). The red coroutine also has found one more red ⦶ state. In Figure 3(c), it has once more found a red ⦶ state; as now the total number of red states is at least nine, which is larger than $\frac{1}{2}|RfnB| = 17/2$, it is aborted. The blue coroutine continued to handle four transitions. In Figure 3(d), the situation immediately before splitting the block is shown. Here also the blue coroutine handled one state without incoming transitions (indicated with "***x***"). In Figure 3(e), the situation at the end of Refine is shown: in the red subblock, old bottom states are marked red ⦶, and new bottom states have been identified (indicated with "nb").



(a) (b) (c) (d) (e)

Fig. 3. Refine extends the red ▥ and blue ▤ markings.

We now consider lines 3.6 to 3.13, used when refining $RedB = split(RfnB, SpB)$ under $\boldsymbol{SpC} \setminus SpB$ in line 2.26. Because we cannot rely on the markings of states here, we look for initial sets of red and blue states differently. Red states are the sources of transitions from *RedB* to $\boldsymbol{SpC} \setminus SpB$, given as parameter *FromRed*. Traversing these transitions requires time proportional to the outgoing transitions of the red states. To determine the initially blue states, we traverse all bottom states that are not yet known to be red (these are the states in the set *Test*) and check whether they have outgoing transitions to $\boldsymbol{SpC} \setminus SpB$. This information was stored with each relevant state in lines 2.15 and 2.18. In addition to traversing the blue bottom states, we spend $\mathcal{O}(Test \setminus Blue)$ time. Observe that all bottom states in $RedB \supseteq Test \setminus Blue$ have been marked before the first call to Refine, so we can traverse them once more without increasing the total time complexity. Other blue (nonbottom) states are found as described before; however, as we might start looking for such blue states before all initially red states have been found because of the balancing of work between the coroutines, we sometimes have to check whether a potentially blue state *s* has a transition to $\boldsymbol{SpC} \setminus SpB$. This is done by traversing the outgoing transitions of *s*. The condition $s' \not\to \boldsymbol{SpC}$ in line 3.23ℓ is checked in another small loop that needs to be taken into account for the time complexity. As it may need $\mathcal{O}(|out(s')|)$ time, we delay it as much as possible and execute it only when



(a) (b) (c) (d) (e)

Fig. 4. Refine also finds the initially red ◍ and blue ⊜ states.

there is no inert transition to a nonblue state left. If no transition to $\boldsymbol{SpC} \setminus SpB$ is found, the state is blue, and the time spent on the test is proportional to its number of outgoing transitions. Otherwise, the state is a new bottom state: it is a red nonbottom state, but all its inert transitions go to blue states. As only nonbottom states undergo this test, the latter outcome will be found at most once, and we consider the time spent on this test to be part of the handling of the new bottom state (item (3) of Theorem 4.4).

To summarize, if the blue subblock is smaller, the refinement uses $\mathcal{O}(|Test \setminus Blue| + |in(NewB)| + |out(NewB)| + |out(NewBott)|)$ time, with $NewBott$ the set of new bottom states; it uses $\mathcal{O}(|in(NewB)| + |out(NewB)|)$ time if the red subblock is smaller, as $|FromRed| \leq |out(NewB)|$. In both cases, it uses time proportional to the number of incoming and outgoing transitions of the smaller resulting block (item (2) of Theorem 4.4) and possibly $\mathcal{O}(|Test \setminus Blue|)$, which is attributed to an earlier state marking, and $\mathcal{O}(|out(NewBott)|)$, which is attributed to the new bottom states.

Example 5.3. Such a call of Refine is illustrated in Figure 4. In Figure 4(a), we see the situation when calling Refine from line 2.26: no states are marked, but every bottom state has a transition to $\boldsymbol{NewC}$. So, it was marked previously, and it is known whether it has a transition to $\boldsymbol{SpC} \setminus SpB$, stored in lines 2.15 and 2.18. The three bottom states are in $Test$.

In Figure 4(b), the situation is shown after each coroutine has taken two steps to find the initially red ◍ and blue ⊜ states: The red coroutine has checked two transitions from $RfnB$ to $\boldsymbol{SpC} \setminus SpB$ and found one red (bottom) state. The blue coroutine has determined that one bottom state has no transition to $\boldsymbol{SpC} \setminus SpB$, but another one has. Therefore, the *blue* coroutine colored the latter state red ◍: the state has to be removed from $Test$ and cannot be added to $Blue$. As now $Test = \emptyset$, the blue coroutine proceeds to handling predecessors of blue states.

In Figure 4(c), it has found a state, indicated with “0,” without any inert transition to nonblue states, but it cannot color this state blue ⊜: Because *FromRed* $\neq \emptyset$, it needs to test slowly whether $s' \not\to \boldsymbol{SpC}$ in line 3.23ℓ by walking through the outgoing (noninert) transitions, and it finds a transition that leads to $\boldsymbol{SpC} \setminus \mathit{SpB}$. The red coroutine, in the meantime, has handled more transitions, but not found more red states.

Finally, in Figure 4(d), the red and blue coroutines have found all their states: The red coroutine empties *FromRed* and finds one more red ⦶ state. The blue coroutine finds the other blue ⊜ states; for the top right state, one can test $s' \not\to \boldsymbol{SpC}$ in line 3.23ℓ quickly because it has a transition to ***NewC***, so it was marked. The top left state was only found after *FromRed* was emptied by the red coroutine, so the test is not necessary. Both coroutines have to do one last step, namely, check whether the last state has incoming inert transitions. Depending on which coroutine goes first, one or the other subblock is moved to *NewB*.

In Figure 4(e), the situation when Refine returns is shown: the nonblue state for which the test $s' \not\to \boldsymbol{SpC}$ was executed slowly has become a new bottom state in the red subblock.

5.3. Handling New Bottom States

As mentioned previously, new bottom states can be introduced in subblock *RedB* by splitting. These are found in lines 3.31 to 3.38. Sometimes, such a new bottom state can no longer reach all the constellations that were reachable from the original bottom states, making the partition unstable under these constellations. In such a case, to re-establish Invariant 4.1, we have to further refine the red subblock.

The extra refinements are executed in a separate procedure PostprocessNewBottom, which is shown in Algorithm 4. We have to test, for every constellation reachable from block *RfnB*, whether there are new bottom states that *cannot* reach it. However, we want to avoid doing useless work in handling old bottom states. We therefore first split *RfnB* into a part that contains the new bottom states and another part containing the old bottom states in line 4.3. The splitter $\mathit{cosplit}(\mathit{RedB}, \mathit{BlueB})$ used here is not yet a constellation. However, as this refinement will have to be executed at some point anyway, it is guaranteed not to separate states that are stuttering equivalent. Close inspection of Algorithm 3 shows that the splitter constellation is only accessed if *FromRed* $\neq \emptyset$, so it is not a problem for this particular call to Refine.

We refine in this manner, since every new bottom state in *RedB* has a transition to *BlueB*, but no old bottom state has such a transition. Therefore, all new bottom states are in $\mathit{split}(\mathit{RedB}, \mathit{BlueB})$, while old bottom states are in $\mathit{cosplit}(\mathit{RedB}, \mathit{BlueB})$. Still, nonbottom states in $\mathit{split}(\mathit{RedB}, \mathit{BlueB})$ with a transition to $\mathit{cosplit}(\mathit{RedB}, \mathit{BlueB})$ actually can reach all relevant constellations via an inert path to some old bottom state in *RedB*, so it is not strictly necessary to stabilize these nonbottom states. The set $\mathit{cosplit}(\mathit{split}(\mathit{RedB}, \mathit{BlueB}), \mathit{cosplit}(\mathit{RedB}, \mathit{BlueB})) = \mathit{cosplit}(\mathit{RedB}, \mathit{cosplit}(\mathit{RedB}, \mathit{BlueB}))$ contains all new bottom states in *RedB* and only those nonbottom states that do not have an inert path to some old bottom state in *RedB*. This is the minimal set of states that we have to stabilize further.

Lemma 5.4. *If one calls* Refine *in line 4.3, with old bottom states as initially red states and new bottom states as initially blue states, the resulting blue subblock is exactly* $\mathit{cosplit}(\mathit{split}(\mathit{RedB}, \mathit{BlueB}), \mathit{cosplit}(\mathit{RedB}, \mathit{BlueB}))$. *Furthermore, this call will not find any new bottom states.*

Proof. It already has been argued that all new bottom states in *RedB* are in $S_{\mathit{RedB}} := \mathit{split}(\mathit{RedB}, \mathit{BlueB})$ and all old bottom states are in $C_{\mathit{RedB}} := \mathit{cosplit}(\mathit{RedB}, \mathit{BlueB})$. Some

ALGORITHM 4: Refine as required by new bottom states, called in lines 2.24 and 2.28

```
4.1  function PostprocessNewBottom($RedB$, $BlueB$)
     {Stabilise the partition for all new bottom states in $RedB$.}
4.2  Create an empty search tree $\mathcal{R}$ of constellations                  } $\mathcal{O}(1)$
4.3  $\langle ResultB, RfnB\rangle$ := Refine($RedB$, $cosplit(RedB, BlueB)$, {old bottom states
                                   $\in RedB$}, $\emptyset$)
4.4  for all constellations $\boldsymbol{C} \notin \mathcal{R}$ reachable from $RfnB$ do          } $\leq |out(NewBott)|$ iter'ns
4.5      Add $\boldsymbol{C}$ to $\mathcal{R}$                                          } $\mathcal{O}(\log n)$
4.6      Register that the transitions $RfnB \to \boldsymbol{C}$ need postprocessing     } $\mathcal{O}(1)$
4.7  end for
4.8  for all bottom states $s \in RfnB$ do                                    }
4.9      Set the current constellation pointer of $s$ to the first constellation it  } $\mathcal{O}(|NewBott|)$
         can reach                                                             }
4.10 end for                                                                  }
4.11 for all constellations $\boldsymbol{SpC} \in \mathcal{R}$ (in order) do                } $\leq |out(NewBott)|$ iter'ns
4.12     for all blocks $\hat{B}$ with transitions to $\boldsymbol{SpC}$ that need postprocessing do }
4.13         Delete $\hat{B} \to \boldsymbol{SpC}$ from the transitions that need postprocessing  } $\mathcal{O}(1)$
4.14         $\langle RedB, BlueB\rangle$ := Refine($\hat{B}$, $\boldsymbol{SpC}$, $\emptyset$, {transitions $\hat{B} \to \boldsymbol{SpC}$})
4.15         for all old bottom states $s \in RedB$ do                           }
4.16             Advance the current constellation pointer of $s$ to the next constellation it can reach } $\mathcal{O}\left(|out(NewBott) \cap \boldsymbol{SpC}|\right)$
4.17         end for                                                           }
4.18         if $RedB$ contains new bottom states then
4.19             $\langle \_, RfnB\rangle$ := Refine($RedB$, $cosplit(RedB, BlueB)$, {old bottom
                                          states $\in RedB$}, $\emptyset$)
4.20             Register that the transitions $RfnB \to \boldsymbol{SpC}$ need postprocessing } $\mathcal{O}(1)$
4.21             Restart the procedure (but keep $\mathcal{R}$), i. e. go to line 4.4
4.22         end if
4.23     end for
4.24     Delete $\boldsymbol{SpC}$ from $\mathcal{R}$                                         } $\mathcal{O}(\log n)$
4.25 end for
4.26 Destroy all temporary data
4.27 return $ResultB$
```

predecessors of old bottom states also end up in C_{RedB}. Therefore, C_{RedB} is a subset of the red subblock resulting from the refinement in line 4.3.

Among the states in S_{RedB}, those that can reach some old bottom state through transitions that are inert in *RedB* can also reach some state in C_{RedB}. Therefore, the red subblock resulting from the refinement also contains $split(S_{RedB}, C_{RedB})$.

We still have to show that there are not more states in the red subblock. The blue subblock resulting from the refinement contains all new bottom states, and whatever the splitter, it must also contain all nonbottom states without an inert path to old bottom states. These are the states in $cosplit(S_{RedB}, C_{RedB})$.

It remains to be shown that this call to Refine does not find additional new bottom states. This holds because all red nonbottom states have an inert transition to some red bottom state; this transition will remain inert. □

It should be noted that splitting *RedB* into S_{RedB} and C_{RedB} could introduce new bottom states that have a direct transition to *BlueB* and a transition to some state in C_{RedB}. Because of the latter transition, these were not bottom states in *RedB* before the splitting. However, the refinement proposed previously is guaranteed to not turn these states into new bottom states, as splitting $split(S_{RedB}, C_{RedB})$ from C_{RedB} is postponed to the moment when *BlueB* is moved to a new constellation.

Having found the subblock *RfnB* that needs stabilization, we add all constellations it can reach to a (sorted) search tree $\mathcal{R}$ of constellations that we have to check and register the transitions that will be involved in these checks in lines 4.4 to 4.7. The time budget (at most $|out(NewBott)|$ iterations) may actually not be met if we only count the new bottom states that *RfnB* currently has, but if some $\boldsymbol{C}$ is added to $\mathcal{R}$

without being reachable from a current (new) bottom state, the refinement under $\boldsymbol{C}$ will definitely create a new bottom state—it is to this new bottom state that we ascribe the work. Note that this requires that we do not repeatedly add the same constellation; therefore, our data structures will need to provide a way to find exactly those $\boldsymbol{C}$s that are not yet added.

After the loop in lines 4.4 to 4.7, we know that for each constellation added to $\mathcal{R}$, there is at least one (future) new bottom state that can reach it—ensuring $|\mathcal{R}| \leq |out(NewBott)|$, which is important for the time bounds; the blocks containing new bottom states do not contain old bottom states; and these blocks only need to be refined under constellations in $\mathcal{R}$.

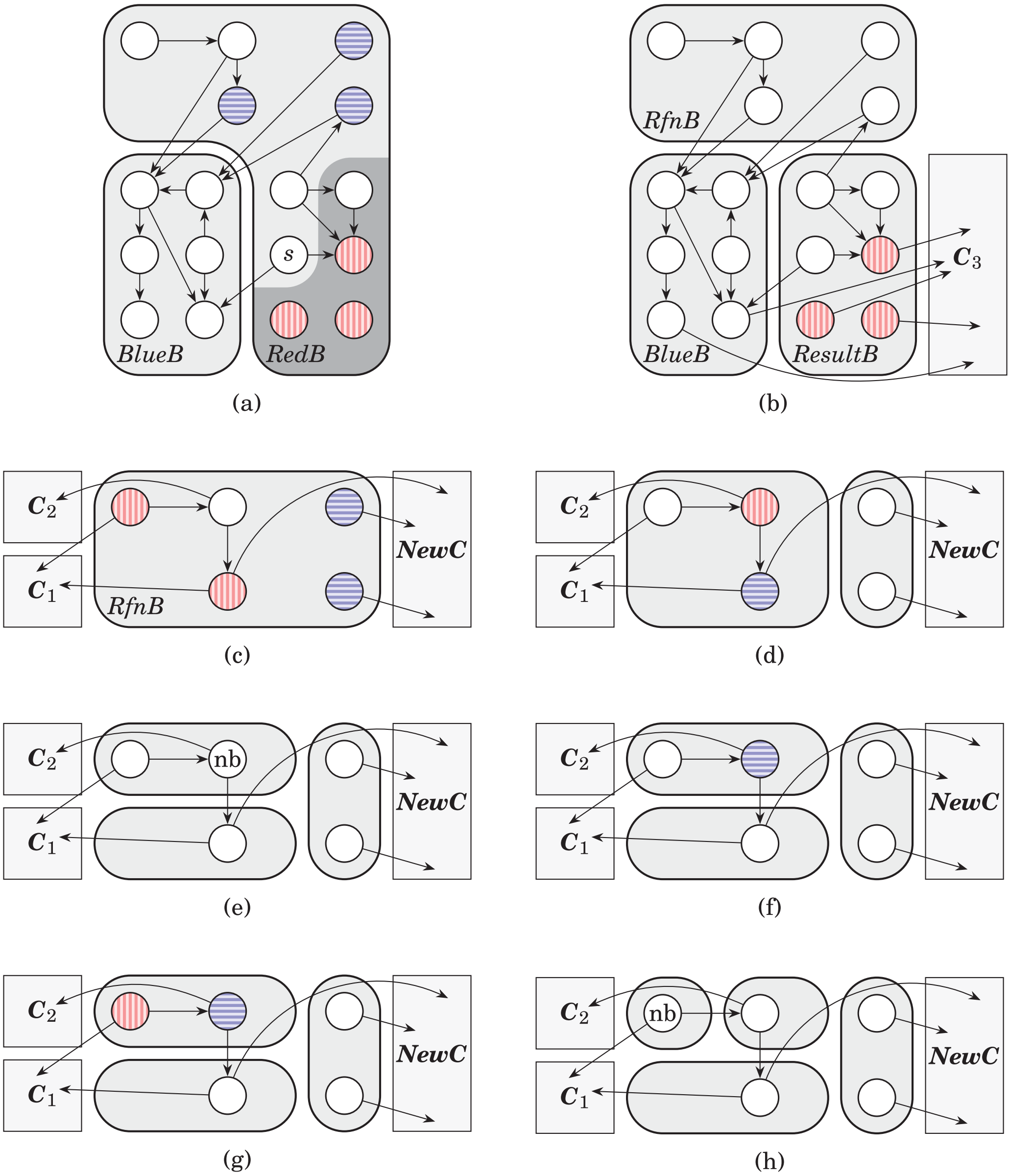
Next, we consider each constellation $\boldsymbol{SpC} \in \mathcal{R}$ in lines 4.11 to 4.25 and refine the relevant blocks under $\boldsymbol{SpC}$. For each $\boldsymbol{SpC} \in \mathcal{R}$, we call Refine for a normal refinement in line 4.14. Because the original *RfnB* may have been refined already, we have to ensure that all its subblocks $\hat{B}$ are refined. Initially, the sources of transitions from $\hat{B}$ to $\boldsymbol{SpC}$ are red, and the bottom states that do not have a transition to $\boldsymbol{SpC}$ are blue. To allow checking for the existence of such a transition from a bottom state to $\boldsymbol{SpC}$ in $\mathcal{O}(1)$ time, we introduce a *current constellation pointer* for each new bottom state b. It always points to the first constellation $\geq \boldsymbol{SpC}$ (in the sort order of $\mathcal{R}$) reachable from b. In other words, a new bottom state is blue if and only if its current constellation pointer does not point to $\boldsymbol{SpC}$. These pointers are initialized in lines 4.8 to 4.10. After line 4.14, the new bottom states that end up in the red subblock need to have their current constellation pointers advanced to the next constellation in lines 4.15 to 4.17, so that these pointers point at the correct constellation the next time those states are visited. Advancing the pointers altogether takes $\mathcal{O}(|out(NewBott)|)$ time.

One call to the blue coroutine of Refine spends $\mathcal{O}(|out(NewBott) \cap \boldsymbol{SpC}|)$ time to skip over the red new bottom states (i.e., $Test \setminus Blue$). As every pair ⟨new bottom state, constellation⟩ is checked once in line 3.8ℓ (when called from line 4.14), one execution of PostprocessNewBottom spends $\mathcal{O}(|out(NewBott)|)$ time to skip over the red new bottom states. The remaining time for refinements is accounted as for refining under other splitters under item (2) of Theorem 4.4.

There is one final complication: during these refinements, we may again find new bottom states. To make sure that the partition is also stable for these new bottom states, we have to backtrack to the beginning in line 4.21 and traverse all constellations reachable from these new bottom states, even if they have been in $\mathcal{R}$ before. During backtracking over such constellations, only the newly detected bottom states need to be handled, so we first split these new bottom states from the ones handled earlier in line 4.19. As mentioned earlier, the loop in lines 4.4 to 4.7 should not run another time over constellations added earlier to $\mathcal{R}$, so that future new bottom states are not causing too much work. Therefore, as $\boldsymbol{SpC}$ is still in $\mathcal{R}$, we have to add the transitions $RfnB \to \boldsymbol{SpC}$ separately in line 4.20 before actually restarting the procedure.

For the elements of $\mathcal{R}$ that have not been checked before backtracking, the newly detected bottom states can be handled together with those found earlier. Consequently, a pair ⟨new bottom state, constellation⟩ is still checked exactly once in line 3.8ℓ (when called from line 4.14), be it during a first call or during backtracking, so not too much time is spent skipping over the red new bottom states.

Example 5.5. Postprocessing is illustrated in Figure 5. It continues where Figure 3 ended. First, in Figure 5(a), the new bottom states are separated from the old bottom states (line 4.3). The part of *RedB* that cannot reach *BlueB* (i.e., the set C_{RedB}) is indicated in dark gray. If we would split *RedB* into S_{RedB} and C_{RedB}, state s would become a new bottom state. We avoid this by adding the states that can reach C_{RedB} to *ResultB*; see Figure 5(b). Additionally, this simplifies the refinement, because it means that we do not have to mark nonbottom states with a transition to *BlueB*, such as s. We



(a) (b) (c) (d) (e) (f) (g) (h)

Fig. 5. PostprocessNewBottom handles new bottom states.

need this first refinement because constellations like $\boldsymbol{C}_3$ in Figure 5(b) would otherwise make postprocessing too slow: there is no new bottom state to which we could ascribe the time spent on (necessarily trivial) refinements under $\boldsymbol{C}_3$.

The rest of PostprocessNewBottom only operates on *RfnB*, the block containing all new bottom states. In this example, we first stabilize under $\boldsymbol{C}_1$ in Figure 5(c). This leads to the blocks drawn in Figure 5(d). Now every block with transitions to $\boldsymbol{C}_2$ (i.e., the left block) is stabilized under $\boldsymbol{C}_2$. This refinement leads to the situation Figure 5(e), in which a new bottom state is found. To this new bottom state we can ascribe the work

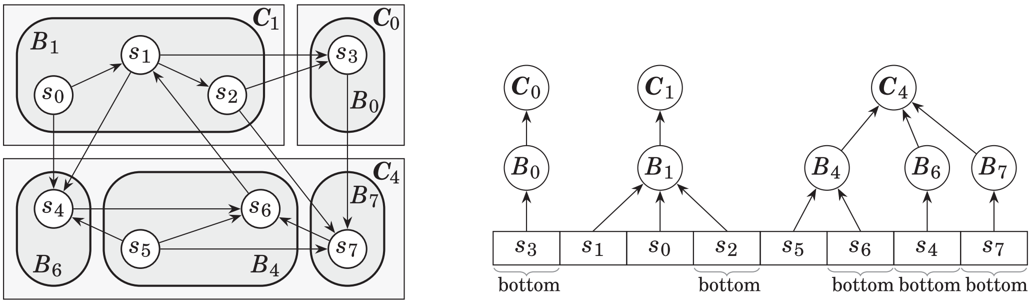


Fig. 6. An example Kripke structure and partition (left) and its refinable partition data structure (right).

for the refinement under $\boldsymbol{C}_2$. Note that in this case, the red subblock does not contain any old bottom states. Because of this new bottom state, PostprocessNewBottom has to backtrack (line 4.18): again, it tries to separate the old from the new bottom states (line 4.19) in Figure 5(f), but obviously the refinement is trivial. Then, this subblock has to be stabilized under $\boldsymbol{C}_1$ again. This is what happens in Figure 5(g). Finally, a one-state block containing yet another new bottom state is produced in Figure 5(h). Before, between, or after the illustrated refinements, these blocks are also stabilized under ***NewC*** and ***SpC***$\setminus$*SpB*. While originally it is superfluous to stabilize under ***NewC***, it may become necessary after some other refinement has found further new bottom states. When postprocessing has finished, *ResultB* is returned, so that it can be further refined in line 2.26, as shown in Figure 4.

6. DATA STRUCTURES

In earlier sections, the algorithm is presented without an explicit reference to the data structures that have been used. Suitable data structures to achieve the required complexity for all basic operations mentioned in the pseudocode are presented in this section. Note that all data structures introduced are proportional either to the number of states or to the number of transitions; hence, the memory complexity of our algorithm is $\mathcal{O}(m)$.

6.1. Data Structure for States and Partitions

To maintain a partition of the states, we use the so-called *refinable partition* data structure [Valmari and Lehtinen 2008]. It contains an array of states that are, in our case, ordered in such a way that first of all, states belonging to the same block are adjacent, and second of all, blocks belonging to the same constellation are adjacent. We call this array **permutation**. In addition, a two-level hierarchy of what we call *descriptors* is used to identify the block and constellation to which a state belongs and, in the other direction, the slice of **permutation** containing the states of a particular block or constellation. Finally, a second array is used to store additional information for each state; unlike **permutation**, the elements in this array always remain in the same order, allowing fast retrieval of information about a particular state. The information for a state s indicates, among other things, the position of s in **permutation**, the identity of the block containing s, and indices indicating where the incoming and outgoing transitions of s are stored. How transitions are stored is explained in the next subsection, and a detailed description of the information stored per state is given in Section 6.3.

Figure 6 shows an example of the use of the refinable partition data structure. Each descriptor for a block or constellation indicates which slice of **permutation** contains its states; each block also indicates which slice of **permutation** contains its bottom states. We place the bottom states always at the end of a block slice.

When a block *RfnB* is split, the new block *NewB* is placed either near the beginning or near the end of *RfnB*, and the states of *NewB* are moved in **permutation** to the beginning or end, respectively, of the states remaining in *RfnB*. The slice that contained *RfnB* as a whole remains in the same position, so the constellation to which *RfnB* belonged now automatically also contains *NewB*. Likewise, when a constellation ***SpC*** is split into ***NewC*** and ***SpC*** $\setminus$ ***NewC***, ***NewC*** is placed either near the beginning or near the end of ***SpC*** in **permutation**.

In the following, we sometimes refer to sorting the constellations in an array or sorting the elements in another array w. r. t. the constellations they refer to. In those cases, we use as sort key the begin index of the slice of the constellation. This implies that, when splitting a constellation as mentioned earlier, ***NewC*** becomes an immediate predecessor or successor of ***SpC*** in the sort order, and the order of other constellations does not change.

When refining, we avoid the need to move states when a splitter is moved to a new constellation by always selecting either the first or the last block in a constellation as the splitter.

6.2. Data Structure for Transitions

To enable fast retrieval, three copies of the transitions are stored in three different arrays.

First, we store the lists of outgoing transitions of each state together in one array **out**; that is, the transitions in this array are sorted by source state. Within each slice of transitions belonging to a source state s, the transitions are sorted according to their target constellation. While refining a constellation, we have to do $\mathcal{O}(|in(\boldsymbol{NewC})| + |out(\boldsymbol{NewC})|)$ work to sort the transitions; this can be done in lines 2.14 and 2.17. For this reason, line 2.17 spends $\mathcal{O}(|out(\boldsymbol{NewC})|)$ time.

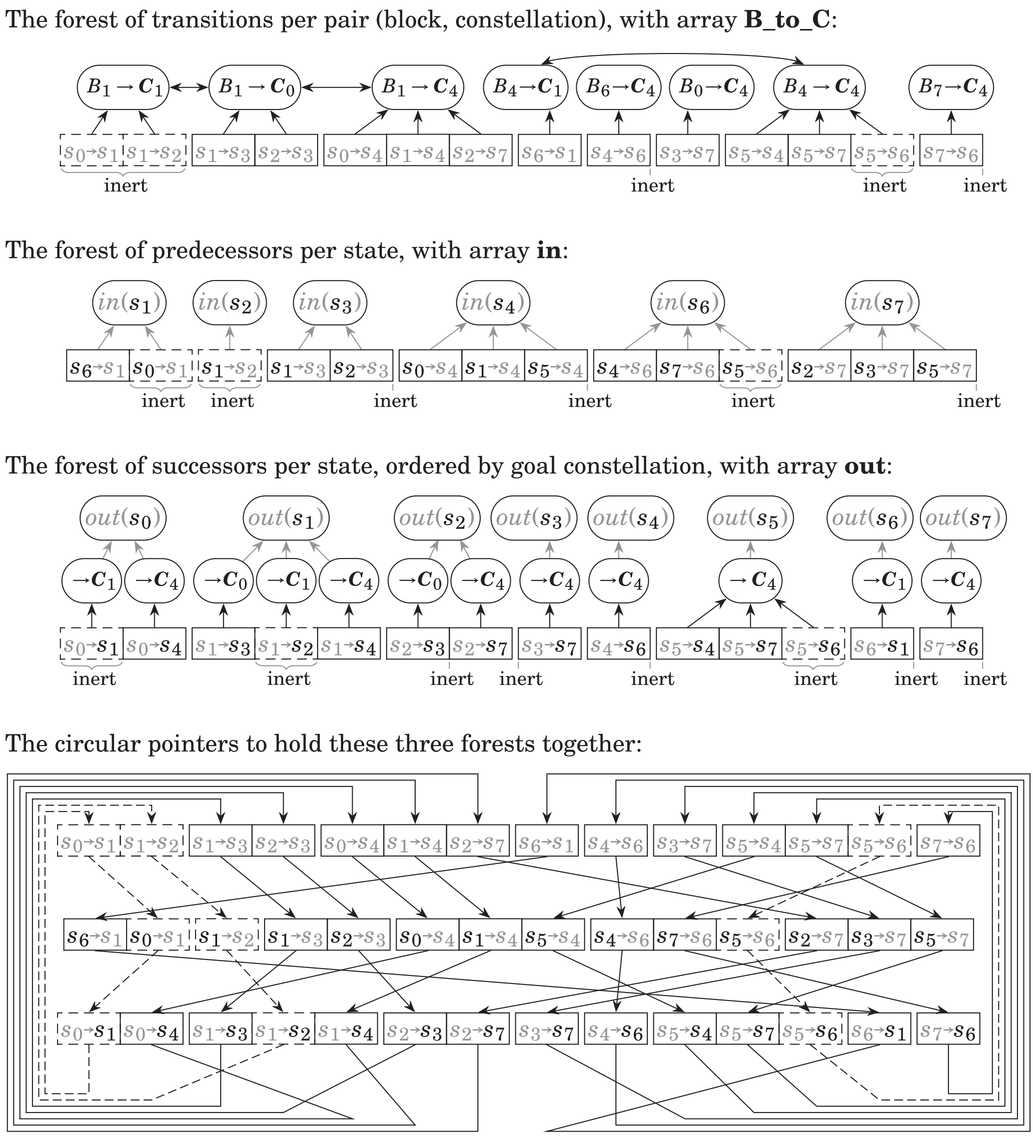
Second, we need to store the lists of incoming transitions of each state, which we do in a similar way to the outgoing transitions in an array **in** (but here, sorting for target state is enough).

Third, the transitions from each block B to constellation $\boldsymbol{C}$ (used as parameter *FromRed* in Refine) are stored in an array **B_to_C** in slices indexed by their ⟨block, constellation⟩ pair. We do not require a particular order in which these slices are stored in **B_to_C**; it is enough to have transitions adjacent that belong to the same ⟨block, constellation⟩ pair. This order of transitions allows splitting slices in **B_to_C** locally whenever one refines a target constellation and when one refines a source block.

The three arrays **out**, **in**, and **B_to_C** need to distinguish noninert and inert transitions; in each slice, we place the inert transitions at the end. We add descriptors to the arrays that indicate their structure: for each source state s and target constellation $\boldsymbol{C}$, there is a descriptor indicating the corresponding slice of **out** containing exactly the subset of transitions in $out(s)$ leading to $\boldsymbol{C}$. Furthermore, for each pair ⟨source block, goal constellation⟩, there is a descriptor of the corresponding slice in **B_to_C**. Further descriptors are stored in the array with additional information for each state.

We require for every block a list of reachable constellations; to this end, we equip the ⟨source block, goal constellation⟩ descriptors with a list structure. During some phases of the algorithm, this list is ordered to store additional information.

All discussed data structures are illustrated in Figure 7 for the same Kripke structure as in Figure 6. The three copies of the transitions should point to each other. As a result, it is not necessary to store data more than once: In each **in** entry, only the source state and a pointer to the corresponding entry in **out** is stored. In each entry of **out**, the target state and a pointer to the corresponding entry in **B_to_C** is stored. Finally, in



The forest of transitions per pair (block, constellation), with array **B_to_C**:

The forest of predecessors per state, with array **in**:

The forest of successors per state, ordered by goal constellation, with array **out**:

The circular pointers to hold these three forests together:

Fig. 7. The data structure for the transitions belonging to the example in Figure 6.

each entry of **B_to_C**, a pointer to the corresponding entry in **in** is needed. The latter two entries also contain pointers to their respective descriptors. The rest of the tree structures do not need to be stored explicitly; the gray arrows in Figure 7 indicate redundant pointers.

6.3. Data Items Stored

We store the following information, where *↗blocktoconst* and *↗statetoconst* are pointers to the descriptors introduced in Section 6.2:

—An instance of the refinable partition data structure with an array **permutation** : *stateposition* → *state*.

- The three transition arrays **out** : *succposition* → *state* × ↗*statetoconst* × *blockposition*, **in** : *predposition* → *state* × *succposition*, and **B_to_C** : *blockposition* → ↗*blocktoconst* × *predposition*.
- For each nonempty set of transitions from block B to constellation $\boldsymbol{C}$, a descriptor indicating the slice of **B_to_C** containing these transitions. These records are designated by the type name *blocktoconst*.
- For each nonempty set of transitions from state s to constellation $\boldsymbol{C}$, a descriptor indicating the slice of **out** containing these transitions. These records are designated by the type name *statetoconst*.
- As additional information for state s:
    - The position in **permutation** where s resides.
    - A pointer to the block it resides in.
    - The slice of **out** containing the successors of s, and a subslice containing the successors reached via inert transitions. If the state has no inert transitions, this is the empty slice at the correct position. For example, in Figure 7, the inert transitions of $s_3 \in \boldsymbol{C}_0$ are sorted before the transition $s_3 \to s_7$ because $s_7 \in \boldsymbol{C}_4$ and $\boldsymbol{C}_0$ is sorted before $\boldsymbol{C}_4$.
    - The slice of **in** containing the predecessors of this state, and a subslice containing the predecessors that reach s via an inert transition.
    - The variable *notblue* (to keep track of *notblue*(s)).
    - A current constellation pointer, as an index in **out**, always set to the boundary between two slices of transitions to a single constellation. This pointer can be used to verify in $\mathcal{O}(1)$ time that s can reach these two constellations and none in between (in sort order). For example, in Figure 7, if the current constellation pointer of s_2 points to the boundary between its transitions to $\boldsymbol{C}_0$ and $\boldsymbol{C}_4$, the pointer can be used to show that s_2 has no transition to $\boldsymbol{C}_1$. On the other hand, if the current constellation pointer of s_0 points to the boundary between its two transitions (to $\boldsymbol{C}_1$ and $\boldsymbol{C}_4$, respectively), it does not help to find out whether s_0 has transitions to $\boldsymbol{C}_0$.
- For each block:
    - The slice of **permutation** containing its elements and some subslices to indicate its marked states, sorted by nonbottom and bottom states.
    - A pointer to the constellation it resides in.
    - For blocks marked as refinable (line 2.12), a pointer to the next refinable block. The pointer is NULL if and only if the block is not refinable.
    - The subslice in **B_to_C** containing the inert transitions of B, that is, the subslice at the end of the slice belonging to $\langle B, \boldsymbol{C}\rangle$ with $B \subseteq \boldsymbol{C}$. If the slice belonging to $\langle B, \boldsymbol{C}\rangle$ only has noninert transitions, this is the empty subslice at the end of that slice. If the block has no transitions to its constellation at all (e.g., block B_0 in Figure 6), this is the empty slice at the very beginning of **B_to_C**.
    - A list of *blocktoconst* descriptors to the transitions reachable from this block.
- For each constellation:
    - The slice of **permutation** containing its elements.
    - For nontrivial constellations, a pointer to the next nontrivial constellation. The pointer is NULL if and only if the constellation is trivial.
    - While postprocessing new bottom states, a slice in **B_to_C** that indicates which transitions need postprocessing.

6.4. Basic Operations and Their Time Complexity

In this section, we list the required basic operations and their respective complexities.

Basic Operations in dbStutteringEquivalence. During execution of Algorithm 2, some states in a refinable block are marked. The marked states are stored in separate subslices of **permutation**; see the top lines of Figure 8 for the order of these subslices.

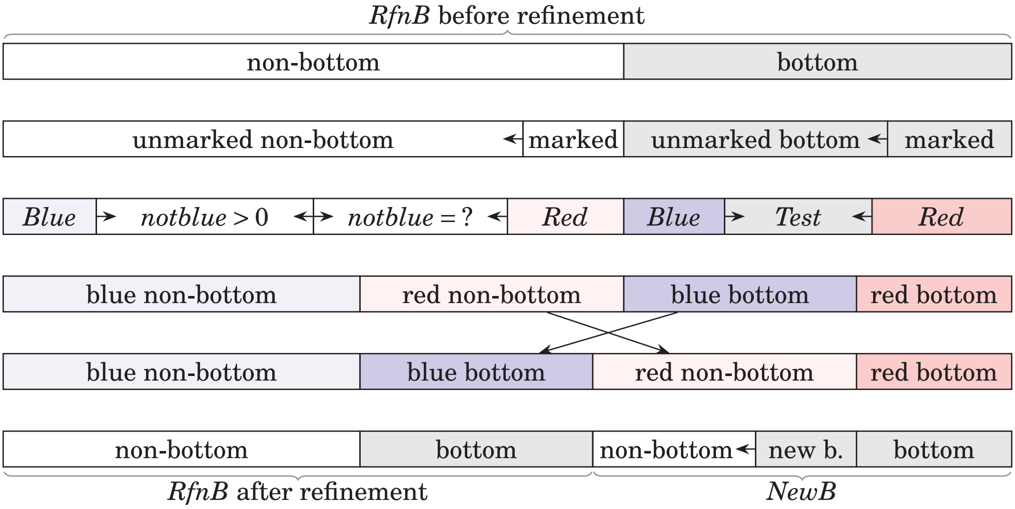


Fig. 8. Internal structure of a block during marking and refinement. In this example, the red subblock is smaller, so it will become the new block.

Operations use the data structures as follows:

—*$\mathcal{C}$ contains a nontrivial constellation* ***SpC***. Check whether the list of nontrivial constellations is nonempty and pick the first element.—$\mathcal{O}(1)$

—*Choose a small splitter block SpB* $\subset$ ***SpC*** *from* $\mathcal{P}$*, that is,* $|SpB| \leq \frac{1}{2}\,|\boldsymbol{SpC}|$. Check the blocks at the beginning and end of the slice of ***SpC*** and pick the smaller one.—$\mathcal{O}(1)$

—*Create a new constellation* ***NewC*** *and move SpB from* ***SpC*** *to* ***NewC***. Allocate a new constellation descriptor; make the constellation pointer of *SpB* point to it; set the slices in ***NewC*** and ***SpC*** accordingly. If necessary, it is possible to mark ***SpC*** as trivial, as ***SpC*** is still the first element in the list of nontrivial constellations.—$\mathcal{O}(1)$

—*$\mathcal{C}$:= partition $\mathcal{C}$ where SpB is removed from* ***SpC*** *...* Nothing needs to be done.

—*Mark a block as refinable.* If the pointer of the block to the next refinable block is NULL, add the block to the respective list.—$\mathcal{O}(1)$

—*Mark a block as nonrefinable.* This operation only needs to be done for the first block in the list of refinable blocks, so the pointer to the head of the list needs to be changed.—$\mathcal{O}(1)$

—*Mark state s' as predecessor.* If s' is in the slice of unmarked nonbottom or bottom states, enlarge the corresponding slice of marked states by decrementing its start index and move s' to that entry.—$\mathcal{O}(1)$ per state.

—*Mark* or *unmark all states of a block as predecessors.* Set the slices of marked states to all elements or to zero elements, respectively.—$\mathcal{O}(1)$

—*Register that $s' \to s$ goes to* ***NewC*** *(instead of* ***SpC****)*. In the correct slice of **B_to_C**, move $s' \to s$ to the beginning of the slice; if necessary, create a new slice descriptor for transitions from the block of s' to ***NewC***; enlarge the slice of transitions to ***NewC*** and reduce the slice of transitions to ***SpC*** by incrementing the end index and start index of the slices, respectively. If the latter slice now has size 0, delete its descriptor. Then, do a similar operation in **out**.—$\mathcal{O}(1)$

—*Store whether s' still has some transition to* ***SpC*** $\setminus$ *SpB*. We reuse the current constellation pointer, introduced originally for postprocessing: Set the current constellation pointer of s to the boundary between ***NewC*** and ***SpC*** $\setminus$ *SpB* in *out*(s'). This boundary is readily available because we just swapped a transition from ***SpC*** to ***NewC***.

Additionally, in the list of constellations reachable from the block of s', move the constellation $\boldsymbol{SpC} \setminus SpB$ to the first place (to prepare it to be used as *FromRed* later on).—$\mathcal{O}(1)$

—*Register that inert transitions from s go to* ***NewC*** *(instead of* ***SpC****)* and *Store whether s still has some transition to* $\boldsymbol{SpC} \setminus SpB$. Repeat the operations mentioned in the previous two items for each inert transition from s in SpB.—$\mathcal{O}(|out_\tau(SpB)|)$

—Refine$(RedB, \boldsymbol{SpC} \setminus SpB, \emptyset, \{transitions\ RedB \to \boldsymbol{SpC} \setminus SpB\})$. In the list of constellations reachable from *RedB*, the first element is $\boldsymbol{SpC} \setminus SpB$. The refinements in lines 2.22 and 4.3 should keep the first item of the list in place. This list element is the descriptor of the desired *FromRed*.—$\mathcal{O}(1)$

Basic Operations in Refine. When a block *RfnB* is refined into red and blue states, we move the red states to the end of its slice in **permutation** and the blue ones to the beginning. See Figure 8 for an illustration of the refining steps. First, the bottom states in *Test* are moved to either *Blue* or *Red* simultaneously; also, some nonbottom states with transitions to *SpB* may already be added to *Red* based on transitions in *FromRed*. Afterward, the nonbottom states are separated. To distinguish between states with defined and undefined *notblue* value (see line 3.19ℓ), we use two separate slices of nonbottom states. After one of the two coroutines has finished, the states are moved into their final positions, and new bottom states are searched in the red subblock.

—***if*** $s \to \boldsymbol{SpC}$. Line 3.8ℓ assumes that the current constellation pointer of s is set to ***SpC*** or to the first constellation sorted after it. This was done in lines 2.15 and 2.18 for the call to Refine in line 2.26 and in lines 4.9 and 4.16 for the call in line 4.14. If this is the case, the test can be executed quickly.—$\mathcal{O}(1)$

—*Move s from Test to Red, Blue* and similar operations. Enlarge the correct slice of *RfnB* by one entry and move s to that entry. If s is a nonbottom state in the slice "$notblue > 0$" and becomes red, a second move is needed to ensure that this slice is not mixed up with the slice "$notblue = ?$".—$\mathcal{O}(1)$

—*Mark s as visited.* Unvisited states should be handled in a specific order, for example, from beginning to end. It is enough to enlarge the correct slice of visited states in *RfnB* by one.—$\mathcal{O}(1)$

—***if***$(\ldots \vee s' \not\to \boldsymbol{SpC})$. Line 3.23$\ell$ tries to use the current constellation pointer of s similar to line 3.8ℓ, but if that is not enough to determine whether $s' \not\to \boldsymbol{SpC}$, we do a binary search through the transitions from s' in **out** to find whether there is a transition to ***SpC***.—$\mathcal{O}(\log |out(s')|)$

—*Move Red* or *Blue to a new block NewB*. This is the moment when the red nonbottom states are swapped with the blue bottom states. Note that some states may stay where they are (in Figure 8, a few red nonbottom states), so the number of swaps is at most $|NewB|$. We also allocate a new block descriptor for *NewB*, change for each state in *NewB* the pointer to its block descriptor, and split the slices in **B_to_C** for these states. During postprocessing, the slice in **B_to_C** for the blue subblock should always be moved toward the end.—$\mathcal{O}(|out(NewB)|)$

—*Destroy all temporary data.* Nothing needs to be done.

—$s \to s'$ *is no longer inert.* In **B_to_C**, **in** and **out**, move transition $s \to s'$ to the beginning of the slice of inert transitions and reduce that slice by incrementing its start index.—$\mathcal{O}(1)$

—*s is a new bottom state.* Move s to the last nonbottom state and enlarge the slice of bottom states by decrementing its start index.—$\mathcal{O}(1)$

—***return*** *…(with old and new bottom states separated).* Before line 3.31, mark the old bottom states in *RedB* and unmark all other states, so the new bottom states will be the ones that are unmarked.—$\mathcal{O}(1)$

Basic Operations in PostprocessNewBottom. During postprocessing, the list of constellations reachable from a block *RfnB* will contain the constellations that are not in $\mathcal{R}$ at the beginning and those in $\mathcal{R}$ at the end. Each refinement (that splits up this list in two) has to maintain that bisection, to ensure that the **for all** loop in lines 4.4 to 4.7 does not use time for constellations that are already in $\mathcal{R}$.

—*Add* $\boldsymbol{C}$ *to* $\mathcal{R}$. In addition to inserting $\boldsymbol{C}$ in the search tree $\mathcal{R}$, we also move the descriptor for $RfnB \to \boldsymbol{C}$ to the end of the list of **B_to_C**-descriptors of *RfnB*.—$\mathcal{O}(\log n)$

—*Delete* ***SpC*** *from* $\mathcal{R}$. Similarly, we also move the descriptor for $RfnB \to \boldsymbol{SpC}$ to the beginning of the list **B_to_C**-descriptors of *RfnB*.—$\mathcal{O}(\log n)$

—*Register that the transitions* $RfnB \to \boldsymbol{SpC}$ *need postprocessing*. In the descriptor of ***SpC***, add a pointer to the slice in **B_to_C** containing these transitions. Later, even though *RfnB* may have been refined, this slice still contains the same transitions. In line 4.20, it may happen that there are already some transitions from other blocks to ***SpC*** that need postprocessing. To ensure the correctness, these transitions should be adjacent to the transitions $RfnB \to \boldsymbol{SpC}$. This is why we move the transitions of the blue subblock in **B_to_C** toward the end.—$\mathcal{O}(1)$

—*Delete* $\hat{B} \to \boldsymbol{SpC}$ *from the transitions that need postprocessing*. If the **for all** loop visits the transitions in the order that they are placed in **B_to_C**, we can reduce the size of the slice of transitions associated with the descriptor of ***SpC*** that need postprocessing by increasing the start index of that descriptor.—$\mathcal{O}(1)$

—*Destroy all temporary data*. This should include unmarking the old bottom states of all blocks involved.—$\mathcal{O}(1)$ per block involved in a refinement.

7. APPLICATION TO BRANCHING BISIMULATION

We show that the algorithm can also be used to determine branching bisimulation with complexity $\mathcal{O}(m(\log|Act| + \log n))$. To obtain this result, we use the transformation from De Nicola and Vaandrager [1995] and Reniers et al. [2014]. Branching bisimulation is typically applied to labeled transition systems (LTSs).

Definition 7.1 (*Labeled Transition System*). A *labeled transition system* is a triple $A = (S, Act, \to)$, where

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

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

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

Definition 7.2 (*Branching Bisimulation*). Consider the labeled transition system $A = (S, Act, \to)$. We call a symmetric relation $R \subseteq S \times S$ a *branching bisimulation relation* if and only if for all $s, t \in S$ such that $s\, R\, t$, the following conditions hold for all actions $a \in Act$: If $s \xrightarrow{a} s'$, then

(1) either $a = \tau$ and $s'\, R\, t$, or
(2) there is a sequence $t \xrightarrow{\tau} \cdots \xrightarrow{\tau} t'$ of (zero or more) τ-transitions such that $s\, R\, t'$ and $t' \xrightarrow{a} t''$ with $s'\, R\, t''$.

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

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

Definition 7.3 (*LTS Embedding*). Let $A = (S, Act, \rightarrow)$ be an LTS. We construct the *embedding of* A to be the Kripke structure $K_A = (S_A, AP, \rightarrow, L)$ as follows:

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

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

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

The following theorem stems from De Nicola and Vaandrager [1995].

THEOREM 7.4. *Let A be an LTS and K_A its embedding. Then two states are branching bisimilar in A if and only if they are divergence-blind stuttering equivalent in K_A.*

If we start out with an LTS with n states and m transitions, then its embedding has at most $n + m$ states and $2m$ transitions. Hence, the algorithm requires $\mathcal{O}(m \log(n + m))$ time. As m is at most $|Act|n^2$, this is also equal to $\mathcal{O}(m(\log|Act| + \log n))$.

Valmari [2009] describes an algorithm to find the (strong) bisimulation equivalence classes of an LTS strictly within time $\mathcal{O}(m \log n)$, which is in contrast to earlier algorithms that depended on $|Act|$; for example, Fernandez [1990] includes a $\mathcal{O}(n\,|Act|)$ term (according to Valmari [2009]). The main idea is to define a splitter not as a block, but as a pair ⟨block, action⟩. This allows to skip refinements for actions that are irrelevant for the current splitter block. We conjecture that a similar construction could be used to make the present algorithm run on LTSs directly within time $\mathcal{O}(m \log n)$.

A final note is that the algorithm can also easily be adapted to determine divergence-sensitive branching bisimulation (dsbb) [De Nicola and Vaandrager 1995] and branching bisimulation with explicit divergence (sometimes referred to as divergence-preserving branching bisimulation, dpbb) by van Glabbeek and Weijland [1996]. In both cases, we amend the translation to a Kripke structure, and we use divergence-blind stuttering equivalence on the resulting Kripke structure. The first, dsbb, can be obtained by simply adding a state with a self-loop and a fresh proposition indicating divergence, and a transition to this state from every state on a τ-loop, similar to the way stuttering equivalence is calculated using divergence-blind stuttering equivalence.

For dpbb, the Kripke structure embedding from Definition 7.3 must be modified to achieve totality by adding a new state with a fresh proposition indicating deadlock, and a transition to this state from any state in the Kripke structure that otherwise would be a deadlock (this is the *deadlock extension* described by van Glabbeek et al. [2009, Section 8]). In addition, divergence must be indicated in the same way as for dsbb.

8. EXPERIMENTS

The new algorithm has been implemented as part of the mCRL2 toolset [Cranen et al. 2013]. The toolset already offered implementations of GV and the algorithm by Blom and Orzan [2003], which distinguishes states by their connection to blocks via their outgoing transitions. We refer to the latter as BO. We have extensively tested the new algorithm by applying it to thousands of randomly generated LTSs and comparing the results with those of the other algorithms. While running these tests, we verified many invariants to guarantee the integrity of the data structures and we checked the

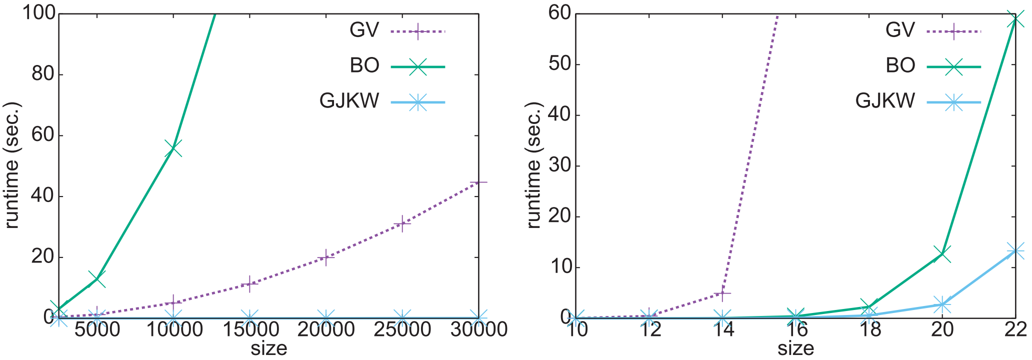


Fig. 9. Running time results for $(a \cdot \tau)^{size}$ sequences (left) and trees of depth *size* (right).

complexity by inspecting that the algorithm never depleted the maximal time budgets for each subtask in the algorithm.

This section reports on our experiments to compare the performance of GV, BO, and the implementation of the new algorithm GJKW.[1] The performance of GV and BO can be very different on concrete examples. All experiments involve the analysis of LTSs, which for GJKW are first transformed to Kripke structures using the translation of Section 7. The reported running times do not include the time to read the input LTS and write the output, but the time it takes to translate the LTS to a Kripke structure and to reduce strongly connected components is included. Groote and Wijs [2016] experimentally compared implementations of GV, BO, and GW. Here, we do not include results obtained with GW, since performance-wise, GW and GJKW are comparable for the selected set of benchmarks. On average, the latter is about 20% faster. When the red and blue coroutines are performed, our GJKW implementation switches between the two after every step. We experienced that other balancing strategies speed up the computation for some cases but slow it down for others.

All experiments have been performed on a machine running CentOS Linux 7.2, with an Intel E5-2698-v3 2.3GHz CPU and 256GB RAM. This machine is part of the DAS-5 cluster [Bal et al. 2016].

Figure 9 presents the running time results for two sets of experiments designed to demonstrate that GJKW has the expected scalability. At the left are the results of analyzing single sequences of the shape $(a \cdot \tau)^n$. As the length $2n$ of such a sequence is increased, the results show that the running times of both BO and GV increase at least quadratically, while the running time of GJKW grows linearly. All algorithms require n iterations, in which BO and GV walk over all the states in the sequence, whereas GJKW only moves two states into a new block.

At the right of Figure 9, the results are displayed of analyzing trees of depth n that up to level $n - 1$ correspond with a binary tree of τ-transitions. Each state at level $n - 1$ has a uniquely labeled outgoing transition to a state at level n. This example is particularly suitable for BO, which obtains the stable partition in a single iteration. Still GJKW beats BO by repeatedly splitting off small blocks of size $2(k - 1)$ if a state at level k is the splitter.

[1] For the experiments, we used version 201409.1.14573M of the mCRL2 toolset available at https://svn.win.tue.nl/repos/MCRL2.

Table I. Running Time and Memory Use Results for GV, BO, and GJKW

	Original		Minimized		Running Time (in s)			Memory Use (in MB)		
Model	n	m	n	m	GV	BO	GJKW	GV	BO	GJKW
vasy_40_60	40,006	60,007	20,003	40,004	24.00	196.00	**0.06**	60	59	70
vasy_18_73	18,746	73,043	2,326	9,751	0.21	0.60	**0.08**	52	56	50
vasy_157_297	157,604	297,000	3,038	12,095	1.60	2.00	**0.30**	103	102	130
vasy_52_318	52,268	318,126	66	333	0.29	1.20	**0.20**	75	98	87
vasy_83_325	83,436	325,584	42,195	197,200	2.40	1.30	**0.60**	110	113	240
vasy_116_368	116,456	368,569	22,398	87,674	0.90	6.00	**0.50**	92	120	130
vasy_720_390	720,247	390,999	3,278	116,537	**0.25**	1.30	0.50	109	116	190
vasy_69_520	69,754	520,633	69,753	520,632	**1.20**	5.00	1.40	145	168	369
cwi_371_641	371,804	641,565	2,134	5,634	6.00	7.00	**1.10**	170	248	200
vasy_166_651	166,464	651,168	42,195	197,200	4.50	3.00	**1.20**	150	165	371
cwi_214_684	214,202	684,419	478	1,612	1.40	13.00	**0.70**	137	186	150
cwi_142_925	142,472	925,429	23	49	1.20	1.40	**0.80**	150	149	160
vasy_386_1171	386,496	1,171,872	71	108	1.40	4.00	**1.08**	222	254	284
vasy_66_1302	66,929	1,302,664	51,128	1,018,692	**2.20**	9.00	3.00	224	334	670
vasy_164_1619	164,865	1,619,204	992	3,456	1.70	12.00	**1.60**	200	297	260
vasy_65_2621	65,537	2,621,480	65,536	2,621,440	80.00	32.00	**7.90**	500	700	1,900
cwi_566_3984	566,640	3,984,157	198	791	7.00	11.00	**5.00**	400	523	558
vasy_1112_5290	1,112,490	5,290,860	265	1,300	8.00	27.00	**7.00**	800	941	980
cwi_2165_8723	2,165,446	8,723,465	4,256	20,880	24.00	180.00	**20.00**	1,400	2,217	2,268
vasy_6120_11031	6,120,718	11,031,292	2,505	5,358	130.00	160.00	**24.00**	2,072	2,163	4,112
vasy_2581_11442	2,581,374	11,442,382	704,737	3,972,600	700.00	230.00	**31.00**	1,612	2,216	4,635
vasy_574_13561	574,057	13,561,040	3,577	16,168	44.00	310.00	**14.00**	1,869	2,114	1,500
vasy_4220_13944	4,220,790	13,944,372	1,186,266	6,863,329	1,200.00	460.00	**38.00**	2,351	2,882	6,633
vasy_4338_15666	4,338,672	15,666,588	704,737	3,972,600	1,800.00	300.00	**41.00**	2,591	2,998	6,634
cwi_2416_17605	2,416,632	17,605,592	730	2,899	30.00	26.00	**19.00**	1,600	2,283	1,748
vasy_6020_19353	6,020,550	19,353,474	256	510	40.00	41.00	**20.00**	2,267	3,147	2,267
vasy_11026_24660	11,026,932	24,660,513	775,618	2,454,834	1,900.00	1,300.00	**68.00**	4,163	5,023	10,760
lift6-final	6,047,527	26,539,368	1,699	9,870	59.00	270.00	**51.00**	3,300	9,200	7,250
vasy_12323_27667	12,323,703	27,667,803	876,944	2,780,022	2,500.00	1,100.00	**77.00**	4,236	5,629	11,930
vasy_8082_42933	8,082,905	42,933,110	290	680	100.00	450.00	**57.00**	7,150	7,254	7,246
cwi_7838_59101	7,838,608	59,101,007	62,031	470,230	260.00	6,500.00	**160.00**	6,956	10,837	16,085
dining_14	18,378,370	164,329,284	228,486	2,067,856	730.00	2,000.00	**490.00**	20,156	28,763	24,470
cwi_33949_165318	33,949,609	165,318,222	12,463	71,466	620.00	5,600.00	**500.00**	22,641	39,742	42,404
1394-fin3	126,713,623	276,426,688	160,258	538,936	68,000.00	10,000.00	**1,000.00**	44,050	82,149	60,235

Table I contains performance measures for minimizing LTSs from the VLTS benchmark set[2] and the mCRL2 toolset.[3] Running times and memory use are the averages over 10 runs and are rounded to significant digits. For each case, the best running time result has been highlighted in bold. Some characteristics of each case are given at the left, namely, the number of states (n) and transitions (m) in the original and the minimized LTS.

The cases prefixed by “cwi” and “vasy” in Table I come from the VLTS benchmark set. The other three cases stem from mCRL2 models distributed with the mCRL2 toolset:

—**lift6-final** is based on an elevator model, extended to six elevators;
—**dining_14** is the dining philosophers model with 14 philosophers;
—**1394-fin3** is the 1394-fin model with three processes and two data elements.

The experiments demonstrate that when also applied to actual state spaces of real models, GJKW generally outperforms the best of the other algorithms, often with a factor of 10 and sometimes with a factor of 100. This difference tends to grow as the LTSs get larger. Note that GJKW’s memory use is sometimes higher than GV’s and BO’s. In particular, it is twice that of GV and BO in a third of the cases, and it is comparable otherwise. This is not surprising given the amount of bookkeeping that GJKW requires.

ACKNOWLEDGMENTS

The authors would like to thank Rob van Glabbeek and Jurriaan Rot for insightful comments and discussions, which helped improve the material presented in this article.

REFERENCES

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA.

Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. 2016. A medium-scale distributed system for computer science research: Infrastructure for the long term. *IEEE Comput.* 49, 5 (May 2016), 54–63. DOI:http://dx.doi.org/10.1109/MC.2016.127

Stefan Blom and Simona Orzan. 2003. Distributed branching bisimulation reduction of state spaces. *Electron. Notes Theor. Comput. Sci.* 80, 1, Special issue: PDMC 2003, Parallel and distributed model checking (2003), 99–113. DOI:http://dx.doi.org/10.1016/S1571-0661(05)80099-4

Stefan Blom and Jaco van de Pol. 2009. Distributed branching bisimulation minimization by inductive signatures. In *Parallel and Distributed Methods in verifiCation (PDMC’09)*, Lubos Brim and Jaco van de Pol (Eds.). Electronic Proceedings in Theoretical Computer Science, Vol. 14. Open Publ. Assoc., 32–46. DOI:http://dx.doi.org/10.4204/EPTCS.14.3

Michael C. Browne, Edmund M. Clarke, and Orna Grümberg. 1988. Characterizing finite Kripke structures in propositional temporal logic. *Theor. Comput. Sci.* 59, 1–2 (1988), 115–131. DOI:http://dx.doi.org/10.1016/0304-3975(88)90098-9

Khrishnendu Chatterjee and Monika Henzinger. 2011. Faster and dynamic algorithms for maximal end-component decomposition and related graph problems in probabilistic verification. In *Proceedings of the 22nd Annual ACM–SIAM Symposium on Discrete Algorithms (SODA’11)*. ACM, New York, 1318–1336. DOI:http://dx.doi.org/10.1137/1.9781611973082.101

Sjoerd Cranen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. de Vink, Wieger Wesselink, and Tim A. C. Willemse. 2013. An overview of the mCRL2 toolset and its recent advances. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS’13)*, Nir Piterman and Scott A. Smolka (Eds.). LNCS, Vol. 7795. Springer, Berlin, 199–213. DOI:http://dx.doi.org/10.1007/978-3-642-36742-7_15 See also www.mcrl2.org.

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

[3] http://www.mcrl2.org.

Sjoerd Cranen, Jeroen J. A. Keiren, and Tim A. C. Willemse. 2012. A cure for stuttering parity games. In *Theoretical Aspects of Computing (ICTAC'12)*, Abhik Roychoudhury and Meenakshi D'Souza (Eds.). LNCS, Vol. 7521. Springer, Heidelberg, 198–212. DOI:http://dx.doi.org/10.1007/978-3-642-32943-2_16

Rocco De Nicola and Frits Vaandrager. 1995. Three logics for branching bisimulation. *J. ACM* 42, 2 (1995), 458–487. DOI:http://dx.doi.org/10.1145/201019.201032

Tom van Dijk and Jaco van de Pol. 2016. Multi-core symbolic bisimulation minimisation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'16)*, Marsha Chechik and Jean-François Raskin (Eds.). LNCS, Vol. 9636. Springer, Berlin, 332–348. DOI:http://dx.doi.org/10.1007/978-3-662-49674-9_19

Jean-Claude Fernandez. 1990. An implementation of an efficient algorithm for bisimulation equivalence. *Sci. Comput. Program.* 13, 2 (1990), 219–236. DOI:http://dx.doi.org/10.1016/0167-6423(90)90071-K

Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. 2012. CADP 2011: A toolbox for the construction and analysis of distributed processes. *Softw. Tools Technol. Transf.* 15, 2, Special issue: TACAS 2011 (2012), 98–107. DOI:http://dx.doi.org/10.1007/s10009-012-0244-z

Rob J. van Glabbeek, Bas Luttik, and Nikola Trčka. 2009. Computation tree logic with deadlock detection. *Logical Methods Comput. Sci.* 5, 4 (Dec. 2009), 1–24. DOI:http://dx.doi.org/10.2168/LMCS-5(4:5)2009

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

Jan Friso Groote and Mohammad Reza Mousavi. 2014. *Modeling and Analysis of Communicating Systems*. MIT Pr., Cambridge, MA.

Jan Friso Groote and Frits Vaandrager. 1990. An efficient algorithm for branching bisimulation and stuttering equivalence. In *Automata, Languages and Programming (ICALP'90)*, M. S. Paterson (Ed.). LNCS, Vol. 443. Springer, Berlin, 626–638. DOI:http://dx.doi.org/10.1007/BFb0032063

Jan Friso Groote and Anton Wijs. 2016. An $O(m \log n)$ algorithm for stuttering equivalence and branching bisimulation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'16)*, Marsha Chechik and Jean-François Raskin (Eds.). LNCS, Vol. 9636. Springer, Berlin, 607–624. DOI:http://dx.doi.org/10.1007/978-3-662-49674-9_40

John E. Hopcroft. 1971. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations*, Z. Kohavi and A. Paz (Eds.). Academic Press, New York, 189–196.

David N. Jansen and Jeroen J. A. Keiren. 2016. *Stuttering Equivalence Is Too Slow!* arXiv E-print 1603.05789. http://arxiv.org/abs/1603.05789.

Paris C. Kanellakis and Scott A. Smolka. 1990. CCS expressions, finite state processes and three problems of equivalence. *Inf. Comput.* 86 (1990), 43–68. DOI:http://dx.doi.org/10.1016/0890-5401(90)90025-D

Saul Kripke. 1963. Semantical considerations on modal logic. *Acta Philosophica Fennica* 16 (1963), 83–94.

Weisong Li. 2009. Algorithms for computing weak bisimulation equivalence. In *3rd International Symposium on Theoretical Aspects of Software Engineering (TASE'09)*, Wei-Ngan Chin and Shengchao Qin (Eds.). IEEE, Los Alamitos, CA, 241–248. DOI:http://dx.doi.org/10.1109/TASE.2009.47

Robin Milner. 1980. *A Calculus of Communicating Systems*. LNCS, Vol. 92. Springer, Berlin. DOI:http://dx.doi.org/10.1007/3-540-10235-3

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

Francesco Ranzato and Francesco Tapparo. 2008. Generalizing the Paige–Tarjan algorithm by abstract interpretation. *Inf. Comput.* 206, 5, Special issue: The 17th International Conference on Concurrency Theory (CONCUR'06) (2008), 620–651. DOI:http://dx.doi.org/10.1016/j.ic.2008.01.001

Michel A. Reniers, Rob Schoren, and Tim A. C. Willemse. 2014. Results on embeddings between state-based and event-based systems. *The Comput. J.* 57, 1 (2014), 73–92. DOI:http://dx.doi.org/10.1093/comjnl/bxs156

Antti Valmari. 2009. Bisimilarity minimization in $O(m \log n)$ time. In *Applications and Theory of Petri Nets (PETRI NETS'09)*, Giuliana Franceschinis and Karsten Wolf (Eds.). LNCS, Vol. 5606. Springer, Berlin, 123–142. DOI:http://dx.doi.org/10.1007/978-3-642-02424-5_9

Antti Valmari and Petri Lehtinen. 2008. Efficient minimization of DFAs with partial transition functions. In *25th International Symposium on Theoretical Aspects of Computer Science (STACS'08)*, Susanne Albers and Pascal Weil (Eds.). LIPIcs, Vol. 1. Schloss Dagstuhl, Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 645–656. DOI:http://dx.doi.org/10.4230/LIPIcs.STACS.2008.1328

Heikki Virtanen, Henri Hansen, Antti Valmari, Juha Nieminen, and Timo Erkkilä. 2004. Tampere verification tool. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, Kurt Jensen and Andreas Podelski (Eds.). LNCS, Vol. 2988. Springer, Berlin, 153–157. DOI:http://dx.doi.org/10.1007/978-3-540-24730-2_12

Thuy Duong Vu. 2007. Deciding orthogonal bisimulation. *Formal Aspects Comput.* 19, 4 (2007), 475–485. DOI:http://dx.doi.org/10.1007/s00165-007-0023-x

Anton Wijs. 2015. GPU accelerated strong and branching bisimilarity checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*. LNCS, Vol. 9035. Springer, Berlin, 368–383. DOI:http://dx.doi.org/10.1007/978-3-662-46681-0_29

Received September 2016; accepted February 2017