A Lazy Data Structure to Accelerate a Recent Algorithm for Branching Bisimulation

David N. Jansen*
Key Laboratory of System Software (Chinese Academy of Sciences)
Institute of Software, Chinese Academy of Sciences, Beijing, China
dnjansen@ios.ac.cn

Jan Friso Groote
Department of Computer Science, Eindhoven University of Technology, The Netherlands
J.F.Groote@tue.nl

Reza Soltani
Formal Methods and Tools, University of Twente, Enschede, The Netherlands
r.soltani@utwente.nl

In [1] we presented a new algorithm to efficiently calculate branching bisimilarity on labelled transition systems. This algorithm relies on a data structure called "BLC sets". However, maintaining this data structure is often not necessary, and some performance improvement can be obtained by calculating the BLC sets in a lazy manner, namely only when they are really needed. Especially, when the algorithm is used to calculate strong bisimilarity, skipping the calculation of BLC sets leads to running time improvement.

1 Introduction

Checking whether processes are equivalent is one way to verify correctness of a system: if a (simple) specification and a (more detailed) implementation are equivalent, one can say that the implementation is correct. The equivalence is an equivalence of behaviours.

Bisimulation is a fundamental notion of behaviour equivalence when behaviour is modelled as LTSs (Labelled Transition Systems). It is an equivalence relation between states of processes with the property that equivalent states have equally-labelled transitions to equivalent successors. To enable the comparison between a specification and an implementation, the bisimulation notion should ignore certain internal transitions, which in LTSs are usually labelled with τ. Branching bisimulation is the variant of bisimulation that treats τ-labelled transitions as invisible and can be computed efficiently.

In this short paper, we are looking at branching bisimulation minimisation, an algorithm to find all equivalent states in an LTS. To compare a specification and an implementation, one may apply branching bisimulation minimisation to the union of the two and check whether their initial states are equivalent. An efficient branching bisimulation minimisation algorithm is also very useful as a preprocessing step for other behaviour-oriented algorithms, e.g., to show that two processes are weak-bisimulation or weak trace equivalent.

In [1] we proposed an efficient algorithm to calculate branching bisimulation with time complexity $O(m \log n)$, where m is the number of transitions and n the number of states of the LTS, based on [8, 2].

*Supported by Beijing Natural Science Foundation Project No. IS25071.

Submitted to:
MISSION@ETAPS

Among the various algorithms for branching bisimulation with time complexity $O(m \log n)$, it is the most efficient both in space and time. There are algorithms whose theoretical time complexity is worse but that appear remarkably efficient in practice, e.g., signature-based algorithms, see for instance [7], although there are typical examples where they perform badly.

Going back to the algorithm in [1], we observed that the so-called BLC sets data structure uses a lot of time to maintain, but it is only used in a few specific situations. This raises the question whether it is possible to improve the practical performance of this algorithm by delaying the calculation of this data structure until it is actually needed. This is the topic of this paper.

On the basis of a standard benchmark set, we find that maintaining the BLC sets in a lazy manner gives often a small advantage when calculating branching bisimulation, although it is also sometimes slightly slower. However, when using the branching bisimulation algorithm to calculate strong bisimulation, by simply taking the internal τ actions as visible, the advantage in time is substantial.

As this paper appears in a workshop to analyze space missions, we modelled a *Ground Segment as a Service (GSaaS)* case study, sourced from Ascentio Technologies S.A. in Argentina [9]. Here, the ground segment contributes to a cloud-based support throughout the mission with planning, telemetry reception, user interaction, payload data handling, and command transmission. GSaaS includes malicious attacks, accidental faults, and defensive measures interacting in one single mission-critical workflow. This aspect makes the benchmark interesting from a verification point of view. The main analysis question is not merely whether the ground segment can fail, but also which defences prevent which attacks or faults from propagating to the top-level failure. With the LTS model, we can use reachability analysis to answer questions like: "Is it possible that a telemetry result is corrupted (the top-level failure) if the software supply chain is attacked, although we have audited third-party software (a certain defence)?" We find that the LTS generated for this model can also be reduced faster using the lazy extension compared to the original algorithm in [1].

2 Branching Bisimulation

We use a *labelled transition system* (LTS) as a model for computational behaviour. It consists of a finite set of states S, a finite set of actions Act including the internal action $\tau \in Act$, and a transition relation $\rightarrow \subseteq S \times Act \times S$. We use n for the number of states $|S|$ and m for the number of transitions $|\rightarrow|$. We assume a fixed LTS $(S, Act, \rightarrow)$ in the text below. We also assume that it does not have many states without transitions, implying $n \in O(m)$.

A relation R between states is a *branching bisimulation* iff it is symmetric and for any states $s, t \in S$ such that $s\,R\,t$ we have: if s has a transition $s \xrightarrow{a} s'$, then t can simulate this transition by some sequence $t = t_0 \xrightarrow{\tau} \cdots \xrightarrow{\tau} t_n \xrightarrow{a} t'$ and $s\,R\,t_n$ and $s'\,R\,t'$. If $a = \tau$, it is also ok for t to simulate $s \xrightarrow{\tau} s'$ by doing nothing, provided $s'\,R\,t$ instead of the above condition. States s and t are *branching bisimilar* iff some branching bisimulation R exists with $s\,R\,t$. Branching bisimilarity is an equivalence relation.

Note that states in a τ-loop are bisimilar. Using a standard algorithm to find strongly connected components, these loops can be detected and removed in linear time $O(m)$. We apply this as a preprocessing step. As a consequence, every τ-path is finite.

2.1 Efficient Partition Refinement Algorithm

Bisimulation minimisation algorithms mostly are *partition refinement algorithms.* These algorithms at least need time in $\Omega(m \log n)$ [4], showing that our algorithm has optimal time complexity. Such an

1: $\mathscr{C} := \{S\}$; Construct the initial partition $\mathscr{B}$ based on actions
2: **while** $\mathscr{B} \neq \mathscr{C}$ **do**
3: Pick a constellation $C \in \mathscr{C} \setminus \mathscr{B}$ and a small block $B \subsetneq C$
4: Divide C into two constellations $C_{\text{sml}} = B$ and $C_{\text{lrg}} = C \setminus B$
5: Split B into three sub-blocks using $B \xrightarrow{\tau} C_{\text{lrg}}$ // $B \xrightarrow{\tau} C_{\text{sml}}$ is inert
6: **for all** BLC sets $B' \xrightarrow{a} C_{\text{sml}}$ (with $B' \not\subseteq C_{\text{sml}}$ or $a \neq \tau$) **do**
7: **if** $B' \subseteq C_{\text{lrg}}$ and $a = \tau$ **then** // $B' \xrightarrow{\tau} C_{\text{lrg}}$ is inert
8: Split B' into three sub-blocks using $B' \xrightarrow{\tau} C_{\text{sml}}$
9: **else if** $(B' \xrightarrow{a} C_{\text{lrg}}) \neq \varnothing$ **then**
10: Split B' into four sub-blocks using $B' \xrightarrow{a} C_{\text{sml}}$ and $B' \xrightarrow{a} C_{\text{lrg}}$
11: Stabilize all NewBotSt sub-blocks completely

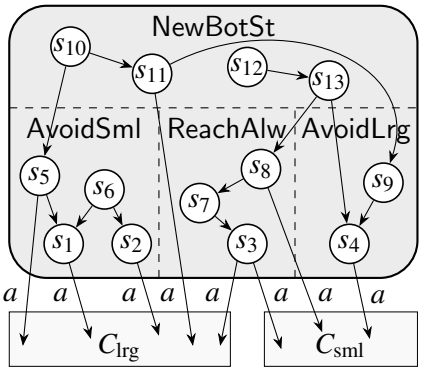

Figure 1: Left: Branching bisimulation algorithm. Right: Example of four-way split.

algorithm maintains a partition $\mathscr{B}$ that represents the current knowledge about branching bisimilarity: states in different blocks are known to be inequivalent.

An important notion is a *bottom state*, which is a state in a block without outgoing τ-transitions within that block. As we removed all τ-loops, each block needs to have at least one bottom state.

The algorithm inspects all blocks to check whether there are still non-branching-bisimilar states in a single block. This is the case when it finds two states s_1, s_2 in a block B of which one has a transition $s_1 \xrightarrow{a} s_1' \in C$ for some block $C \in \mathscr{B}$ that the other cannot simulate, i.e. $s_2 \not\xrightarrow{a} C$. In that case block B is split into parts. Finding blocks that need to be split like B is basically done by going through the transitions backward. The algorithm checks all incoming a-transitions of a certain block C, and if some block B has states with and bottom states without such a transition, B will need to be split. The algorithm of [1] is summarised in Figure 1 left.

To avoid using the same block C repeatedly in this check, one introduces a second partition $\mathscr{C}$ into so-called constellations. Every constellation $C \in \mathscr{C}$ has been a block $\in \mathscr{B}$ at some point, but it may have been split afterwards. Effectively, this means that a constellation is a union of blocks. The main invariant for constellations is that the blocks are stable under the constellations, i.e. if some state in a block B has a non-inert a-transition to a constellation C, then every bottom state in block B has an a-transition to constellation C. In this case a transition $s \xrightarrow{a} t$ is inert iff $a = \tau$ and s and t occur in the same constellation.

If every constellation is a single block, i.e. $\mathscr{B} = \mathscr{C}$, the invariant above implies that $\mathscr{B}$ contains the equivalence classes of a branching bisimulation. Because states in different blocks are inequivalent, $\mathscr{B}$ represents exactly branching bisimulation equivalence.

Otherwise, if $\mathscr{B} \neq \mathscr{C}$, the algorithm divides a constellation $C \in \mathscr{C} \setminus \mathscr{B}$ into two parts: one small block $C_{\text{sml}} := B \subsetneq C$ and the rest $C_{\text{lrg}} := C \setminus B$ (Line 4 in the algorithm in Figure 1). It subsequently restores the invariant by splitting blocks in Lines 5, 8 and 10. The most important split is at Line 10 and splits a block B' with non-inert a-transition to both C_{sml} and C_{lrg} into the following four sub-blocks, illustrated by the example in Figure 1 at the right where unlabelled transitions are τ-transitions:

ReachAlw: states that can always reach both C_{sml} and C_{lrg};
AvoidSml: states that can only reach C_{lrg};
AvoidLrg: states that can only reach C_{sml};
NewBotSt: states that can reach both, but not always (one is reached via AvoidSml or AvoidLrg).

The τ-transitions between C_{sml} and C_{lrg} become non-inert and need to be treated separately in Lines 5 and 8. In both cases, the set of splitting transitions (later, we shall call it the BLC set) is small. Now ReachAlw contains states that can always reach the single target constellation, AvoidSml is empty, and

NewBotSt contains states that can reach the target constellation but also AvoidLrg. Therefore, the split is into three sub-blocks in these two cases.

Part NewBotSt never contains bottom states of block B, but once it has become its own block, it necessarily will contain bottom states, which must be new. In Figure 1 right, states s_{11} and s_{13} become new bottom states because all their τ-transitions leave NewBotSt. These new bottom states do not necessarily satisfy the main invariant, and so this invariant must explicitly be re-established in Line 11. This amounts to splitting NewBotSt further under all outgoing sets of non-inert transitions NewBotSt $\xrightarrow{a}$ C for all $a \in Act$ and $C \in \mathscr{C}$.

The initial $\mathscr{B}$ is constructed in Line 1 by splitting S under all outgoing sets of transitions $S \xrightarrow{a} S$ for $a \neq \tau$ to satisfies the main invariant.

2.2 Optimal Timing

To make the algorithm run efficiently, we use the principle, called "Process the smaller half", by Hopcroft [6]. In our case it says that whenever a state is investigated, it is part of some entity that is at most half of the entity it was part of, the previous time this state was investigated. When we say that a set is *small* we mean it in this sense: at most half the size when it was previously looked into.

The principle "process the smaller half" means that each state can at most be inspected $\log_2 n$ times. We take care that when investigating a state, this state and its incoming and outgoing transition are being manipulated within constant time. Summing up over all states, this gives a time complexity in $O(m \log n)$.

For constellations this means that whenever the algorithm divides a constellation C, C_{lrg} is left alone, and it only works on states and their transitions in C_{sml} where $|C_{\text{sml}}| \leq \frac{1}{2}|C|$.

Similarly, when splitting a block B into parts ReachAlw, ..., NewBotSt, the algorithm should only work proportionally to the parts that are no larger than $\frac{1}{2}|B|$. However, it is not known a priori which part will be too large. Therefore, the algorithm operates on all parts simultaneously and stops to operate on a part that turns out to be larger than $\frac{1}{2}|B|$. It also stops when all but the largest part have been calculated. This largest part is the difference between B and the other parts and therefore, the total time depends only on parts with size $\leq \frac{1}{2}|B|$.

Additionally, we allow a fixed amount of work to be done per state and its transitions at the moment it becomes a new bottom state. This is in particular needed in Line 11, but also to find certain states in NewBotSt. As each state only becomes a bottom state once, this contributes a negligible $O(m)$ to the overall time complexity.

3 BLC Sets And When They Are Needed

The algorithm in [1] heavily relies on so-called BLC sets. A BLC set is the set of transitions from one block B, with one label a, to one constellation C. These sets of transitions can be used to determine how to split block B. The algorithm mentions $B \xrightarrow{\tau} C_{\text{lrg}}$, $B' \xrightarrow{a} C_{\text{sml}}$ and $B' \xrightarrow{a} C_{\text{lrg}}$, which in [1] are all available as BLC sets. In many cases one can avoid constructing them explicitly. Instead of taking $B \xrightarrow{\tau} C_{\text{lrg}}$ and $B' \xrightarrow{a} C_{\text{sml}}$ from the BLC sets, one can go through all outgoing or incoming transitions of $B = C_{\text{sml}}$. However, the BLC sets are still necessary in two situations:

(a) to distinguish AvoidLrg from NewBotSt, if the former is the largest sub-block of B'. Normally, the distinction would work like this. When a potential member of AvoidLrg is found, it is checked whether its outgoing transitions overlap with $B' \xrightarrow{a} C_{\text{lrg}}$, like state s_{11} in Figure 1 right, in time de-

pendent on the transitions of AvoidLrg and of new bottom states (instead of C_{lrg}). But if AvoidLrg is the largest, it is too slow to look for all its potential members, let alone check their transitions.

(b) when handling new bottom states in Line 11, to find under which NewBotSt $\xrightarrow{a} C$ one needs to stabilise. We are allowed to visit the transitions of the (new) bottom states in NewBotSt to find transitions in such BLC sets, but there may also be relevant transitions from states that remain non-bottom. Going through the latter states to find relevant transitions may take too much time if NewBotSt is the largest sub-block of B'.

Both situations only appear in blocks with only new bottom states. In particular, if one uses the algorithm to find strong bisimulation, there are no inert τ-transitions, and therefore, no blocks with new bottom states. In such a case the BLC sets are not necessary.

In the original algorithm of [1], BLC sets are updated upon every split of a constellation or block. Here, we propose to update BLC sets when splitting a constellation as that is helpful in finding all incoming transitions of C_{sml}, grouped by action, but *not* when splitting a block. We call the changed data structure a *super-BLC set*.

In a super-BLC set, transitions from a certain union of blocks, called a *BLC source*, with one label a, to one constellation C are stored. Only in the situations mentioned above, when a BLC set is actually needed, the required block is split off from the respective BLC source. The work to split a BLC source can be assigned to the smaller half, so it fits within time complexity $O(m \log n)$.

Small block counters. Additionally, whenever a block or constellation is split into multiple parts, the algorithm acquires the right to visit all states and transitions in the smaller block one more time. One can keep a counter of how often the states and transitions of a block are allowed to be visited. Sometimes, the use of a BLC set can also be avoided by such a visit, at the cost of decrementing the counter.

Concretely, in situation **(a)** above, if B' is small, it may be more efficient to just check *all* its outgoing transitions if we cannot restrict to the outgoing transitions of potential members of AvoidLrg. In situation **(b)**, splitting a block often produces only a few new bottom states, so NewBotSt is small, and it may be more efficient to run through all its outgoing transitions than to construct its BLC sets.

4 Experimental Evaluation

In order to understand the effect of the lazy BLC optimisation we applied the algorithm to the largest transition systems in the VLTS benchmark set[1] and to an LTS generated from the GSaaS model. The latter LTS is generated by resolving defenses first: First, the model takes visible actions, either $\mathit{enable}(d)$ or $\mathit{skip}(d)$, for each available defense d. Afterward, other actions record the occurrence of basic attacks and faults until the top-level failure action becomes reachable. A state, therefore, summarizes both the active defenses and the attacks/faults that have occurred. This structure is well-suited to the diagnostic question above: from a reduced state space, one can still inspect which defense configurations block which threats. In particular, for branching bisimulation, one can hide the occurrence actions while keeping the defense choices visible, which yields an abstraction that preserves the defense-diagnostic information while abstracting from irrelevant event interleavings.

Table 1 lists the average running time of ten runs to calculate the bisimulation equivalence classes. The numbers are rounded to significant digits; insignificant trailing zeroes are grayed. The two numbers in each benchmark indicate the number of states and transitions divided by thousand. For example, cwi_

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

Benchmark		Strong bisimulation			Branching bisimulation	
$(n/10^3)$	$(m/10^3)$	2025 BB [1]	lazy BLC	Classical BB [5]	2025 BB [1]	lazy BLC
cwi_2165_	8723	3. s	3. s	6. s	4. s	4. s
vasy_6120_	11031	4. s	3. s	20 s	4. s	4. s
vasy_2581_	11442	9. s	6. s	270 s	6. s	6. s
vasy_574_	13561	3. s	1.6s	10. s	4. s	2.1s
vasy_4220_	13944	8. s	5. s	500 s	7. s	6. s
vasy_4338_	15666	11. s	7. s	300 s	8. s	8. s
cwi_2416_	17605	3. s	2.4s	18. s	5. s	5. s
vasy_6020_	19353	7. s	4. s	130 s	1.5s	1.5s
vasy_11026_	24660	15. s	10. s	800 s	16. s	12. s
vasy_12323_	27667	17. s	11. s	1,000 s	19. s	14. s
vasy_8082_	42933	12. s	10. s	20 s	13. s	12. s
cwi_7838_	59101	30 s	20 s	200 s	30 s	40 s
cwi_33949_	165318	130 s	110 s	200 s	120 s	110 s
GSaaS	130734	21. s	15. s	31. s	20 s	19. s
Sum		270 s	200 s	3,500 s	260 s	250 s

Table 1: Performance of branching bisimulation algorithms for strong and branching bisimulation (rounded to significant digits, insignificant zeroes grayed)

2165_8733 has $n \approx 2.165 \cdot 10^6$ states and $m \approx 8.733 \cdot 10^6$ transitions. The GSaaS benchmark extends the evaluation to a larger and structured model with about 100 million states and 130 million transitions.

We applied our branching bisimulation algorithm from 2025 [1] to calculate strong and branching bisimulation, and compared the times to calculate the strong/branching bisimulation equivalence classes with the lazy BLC optimisation described in this paper. As a historical curiosum we also provide the time to calculate strong bisimulation with the classical algorithm for branching bisimulation [5] and observe that modern algorithms perform much better. The benchmarks have been carried out on a Mac Studio with an M1 Ultra processor with the development version of the toolset (version 202507.0.53e2888846M).

We see that the optimisation has a uniform beneficial effect for strong bisimulation, typically reducing the running time up to 30%. Observe that for branching bisimulation the result is mixed, with the majority of cases where the optimisation is beneficial or neutral, but in one case (cwi_7838_59101) the time was increased by roughly 20% (comparing the non-rounded times). Note that the running times to calculate strong and branching bisimulation cannot be compared. Strong bisimulation reduction leads to more equivalence classes than branching bisimulation, requiring more splits. For branching bisimulation splitting a single block is more time-consuming.

Also for GSaaS, the lazy-BLC optimization remains relevant. For strong bisimulation, it still substantially improves over the 2025 implementation. Even for branching bisimulation, the lazy variant gives no penalty. Hence, the Ascentio experiment confirms that the optimization is not limited to one benchmark suite, but also benefits realistic models with long configuration prefixes and many causally independent attack/fault combinations.

5 Conclusion

In the mCRL2 toolset [3] we use branching bisimulation algorithms to calculate strong bisimulation, as this is generally fast enough. In [1] we presented an algorithm that was more space- and time-efficient that all its predecessors. We observed that the algorithm maintained an internal data structure, namely the BLC sets, that were often not used, especially when calculating strong bisimulation. This extended abstract describes how this data structure can be calculated by need, and shows that this is generally beneficial, especially when calculating strong bisimulation.

In the introduction we mentioned the existence of a new paper [7] that indicates that calculating branching bisimulation can be ten times faster for practical benchmarks using so-called signatures, although these algorithms have a much worse time complexity. And indeed on constructed benchmarks, $O(m \log n)$ algorithms as presented here do much better. However, it would be nice to understand why theoretically unattractive algorithms perform better for practical transition systems, which can possibly lead to algorithms that are both theoretically and practically optimal.

References

[1] Jan Friso Groote & David N. Jansen (2025): *A state-based $O(m \log n)$ partitioning algorithm for branching bisimilarity*. In Patricia Bouyer & Jaco van de Pol, editors: *36th international conference on concurrency theory: CONCUR*, *Leibniz international proceedings in informatics (LIPIcs)* 348, Schloss Dagstuhl, Leibniz-Zentrum für Informatik, Dagstuhl, pp. 1–16, doi:10.4230/LIPIcs.CONCUR.2025.18.

[2] Jan Friso Groote, David N. Jansen, Jeroen J.A. Keiren & Anton J. Wijs (2017): *An $O(m \log n)$ algorithm for computing stuttering equivalence and branching bisimulation*. *ACM Trans. Comput. Logic* 18(2):13, doi:10.1145/3060140.

[3] Jan Friso Groote & Jeroen J. A. Keiren (2021): *Tutorial: designing distributed software in mCRL2*. In Kirstin Peters & Tim A. C. Willemse, editors: *Formal techniques for distributed objects, components, and systems: FORTE*, *LNCS* 12719, Springer, Cham, pp. 226–243, doi:10.1007/978-3-030-78089-0_15.

[4] Jan Friso Groote, Jan Martens & Erik P. de Vink (2023): *Lowerbounds for bisimulation by partition refinement*. *Log. Methods Comput. Sci.* 19(2):10, doi:10.46298/LMCS-19(2:10)2023.

[5] Jan Friso Groote & Frits Vaandrager (1990): *An efficient algorithm for branching bisimulation and stuttering equivalence*. In M. S. Paterson, editor: *Automata, languages and programming [ICALP]*, *LNCS* 443, Springer, Berlin, pp. 626–638, doi:10.1007/BFb0032063.

[6] J. Hopcroft (1971): *An $n \log n$ algorithm for minimizing states in a finite automaton*. In Zvi Kohavi & Azaria Paz, editors: *Theory of machines and computations*, Academic Press, New York, pp. 189–196, doi:10.1016/B978-0-12-417750-5.50022-1.

[7] Jan Martens & Maurice Laveaux (2026): *Faster signature refinement for branching bisimilarity minimization*. In Sebastian Junges & Guy Katz, editors: *Tools and algorithms for the construction and analysis of systems: TACAS*, *LNCS* 16505, Springer, Cham, pp. 438–456, doi:10.1007/978-3-032-22752-2_23.

[8] Robert Paige & Robert E. Tarjan (1987): *Three partition refinement algorithms*. *SIAM J. Comput.* 16(6), pp. 973–989, doi:10.1137/0216062.

[9] Reza Soltani, Pablo Diale, Milan Lohupaä-Zwakenberg & Mariëlle Stoelinga (2025): *Safety and security risk mitigation in satellite missions via attack-fault-defense trees*. In Eirik Bjorheim Abrahamsen, Terje Aven, Frederic Bouder, Roger Flage & Marja Ylönen, editors: *35th European safety and reliability conference and 33rd society for risk analysis Europe conference*, Research Publishing, Singapore, pp. 635–642, doi:10.3850/978-981-94-3281-3_ESREL-SRA-E2025-P1598-cd.