Int J Softw Tools Technol Transfer
DOI 10.1007/s10009-017-0468-z

TACAS 2016

Multi-core symbolic bisimulation minimisation

Tom van Dijk[1] · **Jaco van de Pol**[2]

Abstract We introduce parallel symbolic algorithms for bisimulation minimisation, to combat the combinatorial state space explosion along three different paths. Bisimulation minimisation reduces a transition system to the smallest system with equivalent behaviour. We consider strong and branching bisimilarity for interactive Markov chains, which combine labelled transition systems and continuous-time Markov chains. Large state spaces can be represented concisely by symbolic techniques, based on binary decision diagrams. We present specialised BDD operations to compute the maximal bisimulation using signature-based partition refinement. We also study the symbolic representation of the quotient system and suggest an encoding based on representative states, rather than block numbers. Our implementation extends the parallel, shared memory, BDD library Sylvan, to obtain a significant speedup on multi-core machines. We propose the usage of partial signatures and of disjunctively partitioned transition relations, to increase the parallelisation opportunities. Also our new parallel data structure for block assignments increases scalability. We provide SIGREFMC, a versatile tool that can be customised for bisimulation minimisation in various contexts. In particular, it supports models generated by the high-performance model checker LTSMIN, providing access to specifications in multiple formalisms, including process algebra. The extensive experimental evaluation is based on various benchmarks from the literature. We demonstrate a speedup up to 95× for computing the maximal bisimulation on one processor. In addition, we find parallel speedups on a 48-core machine of another 17× for partition refinement and 24× for quotient computation. Our new encoding of the reduced state space leads to smaller BDD representations, with up to a 5162-fold reduction.

Keywords Bisimulation minimisation · Interactive Markov chains · Binary decision diagrams · Parallel algorithms

1 Introduction

One of the main challenges for model checking is that the space and time requirements of model checking algorithms increase exponentially in the size of the models. This paper combines state space reduction, symbolic representation, and parallel computation, to alleviate the state space explosion.

As input models, we consider interactive Markov chains (IMC). These provide a compositional framework to study functionality, performance, and dependability of reactive systems. IMCs inherit non-deterministic choice and communication from labelled transition systems, and probabilistic timed (Markovian) transitions from continuous-time Markov chains.

A state space reduction computes the smallest "equivalent" model. We consider strong bisimilarity, which preserves all behaviour, and branching bisimilarity, which abstracts from internal behaviour (represented by τ-steps) and only preserves the observable behaviour. Note that branching bisimulation preserves the branching structure of an LTS,

Work funded by the NWO Grant 612.001.101 (MaDriD) and by FWF, NFN Grant S11408-N23 (RiSE).

✉ Tom van Dijk
t.vandijk@gmail.com

Jaco van de Pol
J.C.vandePol@utwente.nl

[1] Institute for Formal Methods and Verification, Johannes Kepler University, Linz, Austria

[2] Formal Methods and Tools, University of Twente, Enschede, The Netherlands

Published online: 02 August 2017

thus preserving all properties expressible in CTL*-X [14]. These notions correspond to strong and branching lumping for IMCs.

The reduced state space consists of (representatives of) the equivalence classes in the largest bisimulation, which is typically computed using partition refinement. Starting with the initial partition, in which all states are equivalent, the current partition is refined until the states in any equivalence class can no longer be distinguished. Blom et al. [5] introduced a signature-based method, which defines the equivalence classes according to the characterising signature of a state.

Another important technique to handle large state spaces is symbolic representation. Sets of states are represented by characteristic functions, which are efficiently stored in binary decision diagrams (BDDs). In the literature, symbolic methods have been applied to bisimulation minimisation in several ways. Bouali and De Simone [8] refine the equivalence relation $R \subseteq S \times S$, by iteratively removing all "bad" pairs from R, i.e., pairs of states that are no longer equivalent. For strong bisimulation, Mumme and Ciardo [32] apply saturation-based methods to compute R. Wimmer et al. [40,41] use signatures to refine the partition, represented by the assignment to equivalence classes $P : S \to C$. Symbolic bisimulation based on signatures has also been applied to Markov chains by Derisavi [16] and Wimmer et al. [38,39].

The symbolic representation of the reduced state space tends to be much larger than the original model. One particular application of symbolic bisimulation minimisation is as a bridge between symbolical models and explicit-state analysis algorithms. Symbolical models can have very large state spaces that are efficiently encoded using BDDs. The minimised model has often a sufficiently small number of states, so it can be further analysed efficiently using explicit-state algorithms.

Symbolic techniques mainly reduce the memory requirements of model checking. To speed up the computation, developing scalable parallel algorithms is the way forward, since it takes advantage of multi-core computer systems. In [17,18,20], we implemented the multi-core BDD package Sylvan, providing parallel BDD operations to symbolic model checking.

Parallelisation had been applied to explicit-state bisimulation minimisation before. Blom et al. [4,5] introduced distributed signature-based bisimulation reduction. Also, [29] proposed a concurrent algorithm for bisimulation minimisation which combines signatures with the approach by Paige and Tarjan [33]. Recently, Wijs [37] implemented highly parallel strong and branching bisimilarity checking on GPGPUs. As far as we are aware, no earlier work combines symbolic bisimulation minimisation and parallelism. This paper is an extended version of [21]. There, we demonstrated that specialised BDD operations for signature refinement provide a major speedup of the sequential algorithm, and scale across multiple processors.

We extend [21] by four new results. First, we investigate how to compute the reduced state space, i.e., the quotient of the original system with respect to the maximal bisimulation obtained by signature refinement. Traditionally, the quotient is computed by a sequence of standard BDD operations. Similar to computing the partition, we find that quotient computation benefits from specialised BDD operations. *Second*, we study the representation of the quotient. Traditionally, its states are encoded by using the assigned block number as state identifier. We improve the encoding by choosing one representative state from each block. This considerably reduces the size of the resulting BDD representation. *Third*, we refine our algorithm. Instead of using a monolithic transition relation, we now support a disjunctive partitioning of the transition relation. This appears to be more efficient than a monolithic transition relation and provides further parallelisation opportunities when computing the maximal bisimulation. *Finally*, we link the tool SIGREFMC presented in [21] to LTSMIN, by supporting the partitioned transition systems generated by the symbolic backend of the LTSMIN toolset [6,28,31]. Since LTSMIN supports various input languages, including the specification language MCRL2 [13] for process algebra, this allows us to carry out a considerably larger set of experiments, generated from various specification languages.

Outline This paper presents the following contributions. We recapitulate the notion of partition refinement with partial signatures in Sect. 3. Section 4 discusses how we extended Sylvan to parallelise signature-based partition refinement. In particular, we develop three specialised BDD algorithms: the `refine` algorithm refines a partition according to a signature, but maximally reuses the block number assignment of the previous partition (Sect. 4.3). This algorithm improves the operation cache usage for the computation of the signatures of stable blocks and enables partition refinement with partial signatures. The `inert` algorithm removes all transitions that are not inert (Sect. 4.4). This algorithm avoids an expensive intermediate result reported in the literature [41]. We discuss the new `quotient` computation in Sect. 5. Specialised BDD algorithms significantly speed up the quotient computation for the interactive transition relation (Sect. 5.1) and for the Markovian transition relation (Sect. 5.2). The new encoding of the quotient space is explained in Sect. 5.3. Section 6 presents the implementation of these algorithms as a versatile tool that can be customised for bisimulation minimisation in various contexts, including support for transition systems generated by the model checking toolset LTSMIN (Sect. 6.1). Section 7 discusses experimental data based on benchmarks from the literature. For partition refinement, we demonstrate a speedup of up to 95× sequentially. In addition,

we find parallel speedups of up to 17× due to parallelisation with 48 cores. For quotient computation, we find a speedup of 2–10× by using specialised operations, and we find significantly smaller BDDs (up to 5162× smaller) when using a representative state rather than the block number to encode the new transition system.

2 Preliminaries

We recall the basic definitions of partitions, of labelled transition systems, of continuous-time Markov chains, of interactive Markov chains, and of various bisimulations as in [5,26,40–42].

2.1 Partitions

Definition 1 Given a set S, a partition π of S is a subset $\pi \subseteq 2^S$ such that

$$\bigcup_{C\in\pi} C = S \quad \text{and} \quad \forall C, C' \in \pi : \big(C = C' \vee C \cap C' = \emptyset\big).$$

The elements of π are called equivalence classes or blocks. If π' and π are two partitions, then π' is a refinement of π, written $\pi' \sqsubseteq \pi$, if each block of π' is contained in a block of π. Each equivalence relation $\equiv$ is associated with a partition $\pi = S/\equiv$. In this paper, we use π and $\equiv$ interchangeably.

2.2 Transition systems

Definition 2 A labelled transition system (LTS) is a tuple (S, Act, T), consisting of a set of states S, a set of labels Act, which may contain the non-observable action τ, and transitions $T \subseteq S \times \mathsf{Act} \times S$.

We write $s \xrightarrow{a} t$ for $(s, a, t) \in T$ and $s \xrightarrow{\tau}\!\!\!\!\!\!/\;\;$ when s has no outgoing τ-transitions. We use $\xrightarrow{a*}$ to denote the transitive reflexive closure of $\xrightarrow{a}$. Given an equivalence relation $\equiv$, we write $\xrightarrow[\equiv]{a}$ for $\xrightarrow{a} \cap \equiv$, i.e., transitions between equivalent states, called *inert* transitions. We use $\xrightarrow[\equiv]{a*}$ for the transitive reflexive closure of $\xrightarrow[\equiv]{a}$.

Definition 3 A continuous-time Markov chain (CTMC) is a tuple $(S, \mathbf{R})$, consisting of a set of states S and Markovian transitions $\mathbf{R} : S \to S \to \mathbb{R}_{\geq 0}$.

We write $s \xRightarrow{\lambda} t$ for $\mathbf{R}(s)(t) = \lambda$. The interpretation of $s \xRightarrow{\lambda} t$ is that the CTMC can switch from s to t within d time units with probability $1-\mathrm{e}^{-\lambda\cdot d}$. For a state s, we denote with $\mathbf{R}(s)(C) = \sum_{s'\in C} \mathbf{R}(s)(s')$ the cumulative rate to reach a set of states $C \subseteq S$ from state s in one transition.

Definition 4 An interactive Markov chain (IMC) is a tuple $(S, \mathsf{Act}, T, \mathbf{R})$, consisting of a set of states S, a set of labels Act that may contain the non-observable action τ, transitions $T \subseteq S \times \mathsf{Act} \times S$, and Markovian transitions $\mathbf{R} : S \to S \to \mathbb{R}_{\geq 0}$.

An IMC basically combines the features of an LTS and a CTMC [25,26]. One feature of IMCs is the *maximal progress assumption*. Internal interactive transitions, i.e., τ-transitions, can be assumed to take place immediately, while the probability that a Markovian transition executes immediately is zero. Therefore, we may remove all Markovian transitions from states that have outgoing τ-transitions: $s \xrightarrow{\tau}$ implies $\mathbf{R}(s)(S) = 0$. We call IMCs to which this operation has been applied *maximal-progress-cut* (mp-cut) IMCs. In the rest of this paper, we implicitly assume that IMCs are mp-cut.

2.3 Bisimulation

We recall strong and branching bisimulation. All discussed bisimulations are equivalence relations on the states of a transition system. Two states are bisimilar if and only if there is a bisimulation that relates them. So the maximal bisimulation relates two states if and only if they are bisimilar. For LTSs, we define strong and branching bisimulation as follows [41]:

Definition 5 A strong bisimulation on an LTS is an equivalence relation $\equiv_S$ such that for all states s, t, s' with $s \equiv_S t$ and $s \xrightarrow{a} s'$, there is a state t' with $t \xrightarrow{a} t'$ and $s' \equiv_S t'$.

Definition 6 A branching bisimulation on an LTS is an equivalence relation $\equiv_B$ such that for all states s, t, s' with $s \equiv_B t$ and $s \xrightarrow{a} s'$, either

- $a = \tau$ and $s' \equiv_B t$, or
- there are states t', t'' with $t \xrightarrow{\tau*} t' \xrightarrow{a} t''$ and $t \equiv_B t'$ and $s' \equiv_B t''$.

For CTMCs, we define strong bisimulation as follows [16,38]:

Definition 7 A strong bisimulation on a CTMC is an equivalence relation $\equiv_S$ such that for all $(s, t) \in \equiv_S$ and for all classes $C \in S/\equiv_S$, $\mathbf{R}(s)(C) = \mathbf{R}(t)(C)$.

For mp-cut IMCs, we define strong and branching bisimulation as follows [26,42]:

Definition 8 A strong bisimulation on an mp-cut IMC is an equivalence relation $\equiv_S$ such that for all $(s, t) \in \equiv_S$ and for all classes $C \in S/\equiv_S$,

- $s \xrightarrow{a} s'$ for some $s' \in C$ implies $t \xrightarrow{a} t'$ for some $t' \in C$
- $\mathbf{R}(s)(C) = \mathbf{R}(t)(C)$

Definition 9 A branching bisimulation on an mp-cut IMC is an equivalence relation $\equiv_B$ such that for all $(s,t) \in \equiv_B$ and for all classes $C \in S/\equiv_B$,

- $s \xrightarrow{a} s'$ for some $s' \in C$ implies
  - $a = \tau$ and $(s,s') \in \equiv_B$, or
  - there are states $t', t'' \in S$ with $t \xrightarrow{\tau*} t' \xrightarrow{a} t''$ and $(t,t') \in \equiv_B$ and $t'' \in C$.
- $\mathbf{R}(s)(C) > 0$ implies
  - $\mathbf{R}(s)(C) = \mathbf{R}(t')(C)$ for some $t' \in S$ such that $t \xrightarrow{\tau*} t' \xrightarrow{\tau}\!\!\!\!\!\rightarrow$ and $(t,t') \in \equiv_B$.
- $s \xrightarrow{\tau}\!\!\!\!\!\rightarrow$ implies $t \xrightarrow{\tau*} t' \xrightarrow{\tau}\!\!\!\!\!\rightarrow$ for some t'

As we compare our work to [41,42], we consider divergence-sensitive branching bisimulation for IMCs, which distinguishes deadlock states (without successors) from states that only have self-looping transitions.

3 Signature-based bisimulation minimisation

Blom and Orzan [5] introduced a signature-based approach to compute the maximal bisimulation of an LTS, which was further developed into a symbolic method by Wimmer et al. [41]. Each state is characterised by a *signature*, which is the same for all equivalent states in a bisimulation. These signatures are used to refine a partition of the state space until a fixed point is reached, which is the maximal bisimulation.

In the literature, multiple signatures are sometimes used that together fully characterise states, for example based on the state labels, based on the rates of continuous-time transitions, and based on the enabled interactive transitions. We consider these multiple signatures as elements of a single signature that fully characterises each state.

Definition 10 A signature $\sigma(\pi)(s)$ is a tuple of functions $f_i(\pi)(s)$, that together characterise each state s with respect to a partition π. Two signatures $\sigma(\pi)(s)$ and $\sigma(\pi)(t)$ are equivalent, if and only if for all f_i, $f_i(\pi)(s) = f_i(\pi)(t)$.

The signatures of the five bisimulations from Sect. 2.3 are known from the literature. First, we define for all actions $a \in \mathsf{Act}$ and equivalence classes $C \in \pi$:

- $\mathbf{T}(\pi)(s) = \{(a,C) \mid \exists s' \in C\colon s \xrightarrow{a} s'\}$
- $\mathbf{B}(\pi)(s) = \{(a,C) \mid \exists s' \in C\colon s \xrightarrow[\pi]{\tau*} \xrightarrow{a} s' \wedge \neg(a = \tau \wedge s \in C)\}$
- $\mathbf{R}^s(\pi)(s) = C \mapsto \mathbf{R}(s)(C)$
- $\mathbf{R}^b(\pi)(s) = C \mapsto \max(\{\mathbf{R}(s')(C) \mid \exists s'\colon s \xrightarrow[\pi]{\tau*} s' \xrightarrow{\tau}\!\!\!\!\!\rightarrow\})$

The five bisimulations are associated with the following signatures:

Strong bisimulation for LTS	$(\mathbf{T})$	[41]
Branching bisimulation for LTS	$(\mathbf{B})$	[41]
Strong bisimulation for CTMC	$(\mathbf{R}^s)$	[38]
Strong bisimulation for IMC	$(\mathbf{T}, \mathbf{R}^s)$	[42]
Branching bisimulation for IMC	$(\mathbf{B}, \mathbf{R}^b, s \xrightarrow{\tau*} \xrightarrow{\tau}\!\!\!\!\!\rightarrow)$	[42]

Functions $\mathbf{T}$ and $\mathbf{B}$ assign to each state s all pairs of actions a and equivalence classes $C \in \pi$, such that state s can reach C by an action a either directly ($\mathbf{T}$) or via any number of inert τ-steps ($\mathbf{B}$). Furthermore, inert τ-steps are removed from $\mathbf{B}$. $\mathbf{R}^s$ equals $\mathbf{R}$ but with the domain restricted to the equivalence classes $C \in \pi$ and represents the cumulative rate with which each state s can go to states in C. $\mathbf{R}^b$ equals $\mathbf{R}^s$ for states $s \xrightarrow{\tau}\!\!\!\!\!\rightarrow$ and takes the highest "reachable rate" for states with inert τ-transitions. In branching bisimulation for mp-cut IMCs, the "highest reachable rate" is by definition the rate that all states $s \xrightarrow{\tau}\!\!\!\!\!\rightarrow$ in C have. The element $s \xrightarrow{\tau*} \xrightarrow{\tau}\!\!\!\!\!\rightarrow$ distinguishes time convergent states from time divergent states [42] and is independent of the partition.

For the bisimulations of Definitions 5–9, we state:

Lemma 1 *A partition π is a bisimulation, iff for all s and t that are equivalent in π, $\sigma(\pi)(s) = \sigma(\pi)(t)$.*

For the above definitions, it is fairly straightforward to prove that they are equivalent to the classical definitions of bisimulation. See [5,41] for the bisimulations on LTSs and [42] for the bisimulations on IMCs.

3.1 Signature-based partition refinement

As discussed above, signatures can consist of multiple elements. We first define partition refinement using the full signature. We then define partition refinement with partial signatures, i.e., using the elements of the signature, and discuss advantages of this approach.

Definition 11 (*Partition refinement with full signatures*)

$$\mathrm{sigref}(\pi,\sigma) := \{\{t \in S \mid \sigma(\pi)(s) = \sigma(\pi)(t)\} \mid s \in S\}$$

For a given signature σ, we define the series of partition refinements:

$$\pi^0 := \{S\}$$
$$\pi^{n+1} := \mathrm{sigref}(\pi^n, \sigma)$$

The algorithm iteratively refines the initial coarsest partition $\{S\}$ according to the signatures of the states, until some fixed point $\pi^{n+1} = \pi^n$ is obtained. For monotone signatures (defined below), this fixed point is the maximal bisimulation.

Definition 12 A signature is monotone if for all π, π' with $\pi \sqsubseteq \pi', \sigma(\pi)(s) = \sigma(\pi)(t)$ implies $\sigma(\pi')(s) = \sigma(\pi')(t)$.

For all monotone signatures, the sigref operator is monotone: $\pi \sqsubseteq \pi'$ implies $\text{sigref}(\pi, \sigma) \sqsubseteq \text{sigref}(\pi', \sigma)$. Hence, following Kleene's fixed point theorem, the procedure above reaches the greatest fixed point.

In Definition 11, the full signature is computed in every iteration. We propose to apply partition refinement using parts of the signature. By definition, $\sigma(\pi)(s) = \sigma(\pi)(t)$ if and only if for all parts $f_i(\pi)(s) = f_i(\pi)(t)$.

Definition 13 (*Partition refinement with partial signatures*)

$$\begin{aligned} \text{sigref}(\pi, f_i) &:= \{\{t \in S \mid f_i(\pi)(s) = f_i(\pi)(t) \wedge \\ &\qquad s \equiv_\pi t\} \mid s \in S\} \\ \pi^0 &:= \{S\} \\ \pi^{n+1} &:= \text{sigref}(\pi^n, f_i) \qquad (\text{select } f_i \in \sigma) \end{aligned}$$

We always select some f_i that refines the partition π. A fixed point is reached only when no f_i refines the partition further: $\forall f_i \in \sigma\colon \text{sigref}(\pi^n, f_i) = \pi^n$. The extra clause $s \equiv_\pi t$ ensures that every application of sigref refines the partition.

Theorem 1 *If all parts f_i are monotone, Definition 13 yields the greatest fixed point.*

Proof The procedure terminates since the chain is decreasing ($\pi^{n+1} \sqsubseteq \pi^n$), due to the added clause $s \equiv_\pi t$. We reach some fixed point π^n, since $\text{sigref}(\pi^n, \sigma) = \pi^n$ is implied by $\forall f_i \in \sigma\colon \text{sigref}(\pi^n, f_i) = \pi^n$. Finally, to prove that we get the *greatest* fixed point, assume there exists another fixed point $\xi = \text{sigref}(\xi, \sigma)$. Then, also $\xi = \text{sigref}(\xi, f_i)$ for all i. We prove that $\xi \sqsubseteq \pi^n$ by induction on n. Initially, $\xi \sqsubseteq S = \pi^0$. Assume $\xi \sqsubseteq \pi^n$, then for the selected i, $\xi = \text{sigref}(\xi, f_i) \sqsubseteq \text{sigref}(\pi^n, f_i) = \pi^{n+1}$, using monotonicity of f_i.

There are several advantages to this approach due to its flexibility. First, for any f_i that is independent of the partition, we need to refine with respect to that f_i only once. Furthermore, refinements can be applied according to different strategies. For instance, for the strong bisimulation of an mp-cut IMC, one could refine w.r.t. $\mathbf{T}$ until there is no more refinement, then w.r.t. $\mathbf{R}^s$ until there is no more refinement, then repeat until neither $\mathbf{T}$ nor $\mathbf{R}^s$ refines the partition. Finally, computing the full signature is the most memory-intensive operation in symbolic signature-based partition refinement. If the partial signatures are smaller than the full signature, then larger models can be minimised.

4 Symbolic signature refinement

This section describes the parallel decision diagram library Sylvan, followed by the (MT)BDDs and (MT)BDD operations required for signature-based partition refinement. We describe how we encode partitions and signatures for signature-based partition refinement. We present a new parallelised `refine` function that maximally reuses block numbers from the old partition. Finally, we present a new BDD algorithm that computes inert transitions, i.e., restricts a transition relation such that states s and s' are in the same block.

4.1 Decision diagram algorithms in Sylvan

In symbolic model checking [11], sets of states and transitions are represented by their characteristic function, rather than stored individually. With states described by N Boolean variables, a set $S \subseteq \mathbb{B}^N$ can be represented by its characteristic function $f\colon \mathbb{B}^N \to \mathbb{B}$, where $S = \{s \mid f(s)\}$. Binary decision diagrams (BDDs) are a concise and canonical representation of Boolean functions [10].

An (ordered) BDD is a directed acyclic graph with leaves 0 and 1. Each internal node has a variable label x_i and two outgoing edges labelled 0 and 1. Variables are encountered along each path according to a fixed variable ordering. Duplicate nodes and nodes with two identical outgoing edges are forbidden. It is well known that for a fixed variable ordering, every Boolean function is represented by a unique BDD.

In addition to BDDs with leaves 0 and 1, multi-terminal binary decision diagrams have been proposed [2,12] with leaves other than 0 and 1, representing functions from the Boolean space $\mathbb{B}^N$ onto any set. For example, MTBDDs can have leaves representing integers (encoding $\mathbb{B}^N \to \mathbb{N}$), floating-point numbers (encoding $\mathbb{B}^N \to \mathbb{R}$), and rational numbers (encoding $\mathbb{B}^N \to \mathbb{Q}$). Partial functions are supported using a leaf $\perp$.

Sylvan [17,18,20] implements parallelised operations on decision diagrams using parallel data structures and work-stealing. Work-stealing [7,19] is a load balancing method for task-based parallelism. Recursive operations, such as most BDD operations, implicitly form a tree of tasks. Independent subtasks are stored in queues and idle processors steal tasks from the queues of busy processors.

See Algorithm 1 for a generic example of a BDD operation. This algorithm takes two inputs, the BDDs x and y, to which a binary operation F is applied. Most decision diagram operations first check if the operation can be applied immediately to x and y (line 2). This is typically the case when x and y are leaves. Often there are also other trivial cases that can be checked first. We then consult the operation cache (line 4) to see if this (sub)operation has been computed earlier. The operation cache is required to reduce the time complexity of BDD operations from exponential to polynomial in the size

```
1  def apply(x, y, F):
2      if x and y are leaves or trivial : return F(x, y)
3      Normalise/simplify parameters
4      if result ← cache[(x, y, F)] : return result
5      v = topVar(x,y)
6      do in parallel:
7          low ← apply(x_{v=0}, y_{v=0}, F)
8          high ← apply(x_{v=1}, y_{v=1}, F)
9      result ← lookupBDDnode(v, low, high)
10     cache[(x, y, F)] ← result
11     return result
```

Algorithm 1 Example of a parallelised BDD algorithm: apply a binary operator F to BDDs x and y.

of the BDDs. Sylvan uses a single shared unique table for all BDD nodes and a single shared operation cache for all operations.

Often, the parameters of an operation can be normalised in some ways to increase the cache efficiency. For example, $a \wedge b$ and $b \wedge a$ are the same operation. In that case, normalisation rules can rewrite the parameters to some standard form in order to increase cache utilisation, at line 3. A well-known example is the if-then-else algorithm, which rewrites using rewrite rules called "standard triples" as described in [9].

If x and y are not leaves and the operation is not trivial or in the cache, we use `topVar` (line 5) to determine the first variable of the root nodes of x and y. If x and y have a different variable in their root node, `topVar` returns the first one in the variable ordering. We then compute the recursive application of F to the cofactors of x and y with respect to variable v at lines 7–8. We write $x_{v=i}$ to denote the cofactor of x where variable v takes value i. Since x and y are ordered according to the same fixed variable ordering, we can easily obtain $x_{v=i}$. If the root node of x is on the variable v, then $x_{v=i}$ is obtained by following the low ($i = 0$) or high ($i = 1$) edge of x. Otherwise, $x_{v=i}$ equals x. After computing the suboperations, we compute the result by either reusing an existing or creating a new BDD node (line 9).

Operations on decision diagrams are typically recursively defined on the structure of the inputs. To parallelise the operation in Algorithm 1, the two independent suboperations at lines 7–8 are executed in parallel using work-stealing. To obtain high performance in a multi-core environment, the data structures for the BDD node table and the operation cache must be highly scalable. Sylvan implements several non-blocking data structures to enable good speedups [17, 20].

To compute symbolic signature-based partition refinement, several basic operations must be supported by the BDD package (see also [41]). Sylvan implements basic operations such as $\wedge$ and `if-then-else`, and existential quantification $\exists$. Negation $\neg$ is performed in constant time using complement edges. To compute relational products of transition systems, there are operations `relnext` (to compute successors) and `relprev` (to compute predecessors and to concatenate relations), which combine the relational product with variable renaming. Similar operations are also implemented for MTBDDs. Sylvan is designed to support custom BDD algorithms. We present several new algorithms below.

4.2 Encoding of signature refinement

We implement symbolic signature refinement similar to [41]. However, we do not refine the partition with respect to a single block, but with respect to all blocks simultaneously. We use a binary encoding with variables s for the current state, s' for the next state, a for the action labels, and b for the blocks. We order BDD variables a and b after s and s', since this is required to efficiently replace signatures (on a and b) by new block numbers b (see below). Variables s and s' are interleaved, which is a common heuristic for transition systems.

In [21], we ordered a before b. However, we expect that in general ordering b before a is better for the following reason. If we have a before b, then when computing the signatures and the quotient (Sect. 5), it is guaranteed that all BDD nodes on a variables have to be recreated, whereas they may be reused if a variables are last in the ordering.

To perform symbolic bisimulation, we represent a number of sets by their characteristic functions. See also Fig. 1.

- A set of states is represented by a BDD $\mathcal{S}(s)$;
- Transitions are represented by a BDD $\mathcal{T}(s, s', a)$;
- Markovian transitions are represented by an MTBDD $\mathcal{R}(s, s')$, with leaves containing rational numbers ($\mathbb{Q}$) that represent the transition rates;
- Signatures $\mathbf{T}$ and $\mathbf{B}$ are represented by a BDD $\sigma_T(s, b, a)$;
- Signatures $\mathbf{R}^s$ and $\mathbf{R}^b$ are represented by an MTBDD $\sigma_R(s, b)$, with leaves containing rational numbers ($\mathbb{Q}$) that represent the rates in the signature.

We represent Markovian transitions using rational numbers, since they offer better precision than floating-point numbers. The manipulation of floating-point numbers typically introduces tiny rounding errors, resulting in different results of similar computations. This significantly affects bisimulation reduction, often resulting in finer partitions than the maximal bisimulation [38], which is unacceptable.

In the literature, three methods have been proposed to represent the partition π.

1. As an equivalence relation, using a BDD $\mathcal{E}(s, s') = 1$ iff $s \equiv_\pi s'$ [8,32].
2. As a partition, by assigning each block a unique number, encoded with variables b, using a BDD $\mathcal{P}(s, b) = 1$ iff $s \in C_b$ [16,41,42].

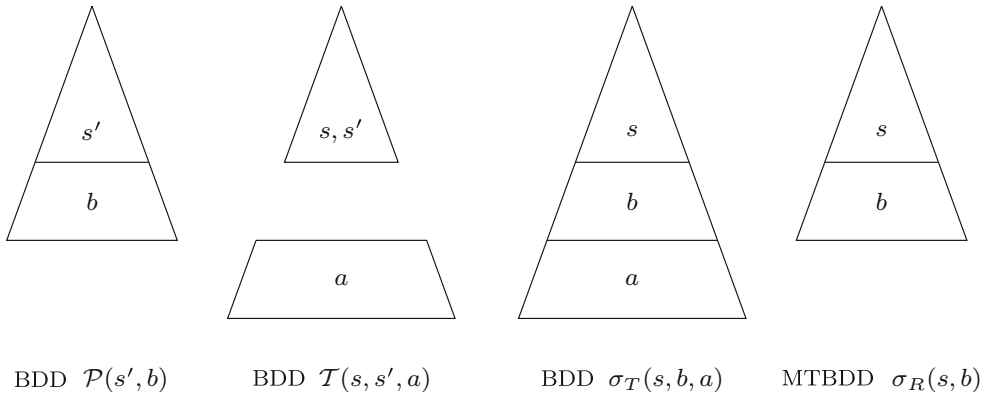

Fig. 1 Schematic overview of the BDDs in signature refinement

3. Using $k = \lceil \log_2 n \rceil$ BDDs $\mathcal{P}_0, \ldots, \mathcal{P}_{k-1}$ such that $\mathcal{P}_i(s) = 1$ iff $s \in C_b$ and the ith bit of b is 1. This requires significant time to restore blocks for the refinement procedure, but can require less memory [15].

We choose to use method 2, since in practice the BDD of $\mathcal{P}(s, b)$ is smaller than the BDD of $\mathcal{E}(s, s')$. Using $\mathcal{P}(s, b)$ also has the advantage of straightforward signature computation. The logarithmic representation is incompatible with our approach, since we refine all blocks simultaneously. Their approach involves restoring individual blocks to the $\mathcal{P}(s, b)$ representation, performing a refinement step, and compacting the result to the logarithmic representation. Restoring all blocks simply computes the full $\mathcal{P}(s, b)$.

In the implementation of signature refinement, we actually encode $\mathcal{P}$ using s' variables instead of s variables, i.e., encoding from target states to block numbers. This is advantageous for signature computation, as the signatures σ_T and σ_R can then be computed as follows:

- $\sigma_T(s, b, a) := \exists s' : \mathcal{T}(s, s', a) \wedge \mathcal{P}(s', b)$
- $\sigma_R(s, b) := \exists_{\mathtt{sum}}\, s' : \mathcal{R}(s, s') \wedge \mathcal{P}(s', b)$

4.3 The `refine` algorithm

We present a new BDD algorithm to refine partitions according to a signature, which maximally preserves previously assigned block numbers.

Partition refinement consists of two steps: computing the signatures and computing the next partition. Given the signatures σ_T and/or σ_R for the current partition π, the new partition can be computed as follows.

Since the chosen variable ordering has variables s, s' before a, b, each path in σ ends in a (MT)BDD representing the signature for the states encoded by that path. For σ_T, every path that assigns values to s ends in a BDD on a, b. For σ_R, every path that assigns values to s ends in a MTBDD on b with rational leaves.

```
1  def refine(σ, P):
2      if result ← cache[(σ, P, iter)] : return result
3      v = topVar(σ, P)      # interpret s' in P as s
4      if v equals s_i for some i :
           # match state in σ and P
5          do in parallel:
6              low ← refine(σ_{s_i=0}, P_{s'_i=0})
7              high ← refine(σ_{s_i=1}, P_{s'_i=1})
8          result ← lookupBDDnode(s'_i, low, high)
9      else:
           # σ now encodes the state signature
           # P now encodes the previous block
10         B ← decodeBlock(P)
           # try to claim block B if still free
11         if blocks[B].sig = ⊥ :
12             cas(blocks[B].sig, ⊥, σ)
13         if blocks[B].sig = σ :
14             result ← P
15         else:
16             B ← search_or_insert(σ, B)
17             result ← encodeBlock(B)
18     cache[(σ, P, iter)] ← result
19     return result
```

Algorithm 2 `refine`, the (MT)BDD operation that assigns block numbers to signatures, given a signature σ and the previous partition $\mathcal{P}$.

Wimmer et al. [41] present a BDD operation `refine` that "replaces" these sub-(MT)BDDs by the BDD representing a unique block number for each distinct signature. The result is the BDD of the next partition. They use a global counter and a hash table to associate each signature with a unique block number. This algorithm has the disadvantage that block number assignments are unstable. There is no guarantee that a stable block has the same block number in the next iteration. This has implications for the computation of the new signatures. When the block number of a stable block changes, cached results of signature computation in earlier iterations cannot be reused.

We modify the `refine` algorithm to use the current partition to reuse the previous block number of each state. This also allows refining a partition with respect to only a part of the signature, as described in Sect. 3. The modification is applied such that it can be parallelised in Sylvan. See Algorithm 2.

The algorithm has two input parameters: σ which encodes the (partial) signature for the current partition and $\mathcal{P}$ which encodes the current partition. The algorithm uses a global counter iter, which is the current iteration. This is necessary since the cached results of the previous iteration cannot be reused. It also uses and updates an array `blocks`, which contains the signature of each block in the new partition. This array is cleared between iterations of partition refinement.

The implementation is similar to other BDD operations, with an operation cache (lines 2 and 18) and a recursion step for variables in s (lines 3–8). The two recursive operations are executed in parallel. `refine` simultaneously descends in σ and $\mathcal{P}$ (lines 6–7), matching the valuation of s_i in σ and s'_i in $\mathcal{P}$. Block assignment happens at lines 11–17. We rely on the well-known atomic operation `compare_and_swap` (`cas`), which atomically compares and modifies a value in memory. This is necessary for parallel correctness. We use `cas` to claim the previous block number for the signature (line 12). If the block number is already claimed for a different signature, then the current block is being split and we call `search_or_insert` to assign a new block number.

Different implementations of `search_and_insert` are possible. We implemented a parallel hash table that uses a global counter for the next block number when inserting a new pair (σ, B), similar to [41]. We also implemented an alternative implementation that integrates the `blocks` array with a skip list. A skip list is a probabilistic multi-level ordered linked list. See [35]. This implementation performed better in our experiments, but we omit the implementation details due to space constraints.

4.4 Computing inert transitions

To compute the set of inert τ-transitions for branching bisimulation $s \xrightarrow{\tau}_{\pi} s'$, or more generally, to compute any inert transition relation $\rightarrow \cap \equiv$ with $\pi = S/\equiv$ with blocks b, the expression $\mathcal{T}(s, s') \wedge \exists b\colon \mathcal{P}(s, b) \wedge \mathcal{P}(s', b)$ must be evaluated. [41] writes that the intermediate BDD of $\exists b\colon \mathcal{P}(s, b) \wedge \mathcal{P}(s', b)$, obtained by first computing $\mathcal{P}(s, b)$ using variable renaming from $\mathcal{P}(s', b)$ and then $\exists b\colon \mathcal{P}(s, b) \wedge \mathcal{P}(s', b)$ using `and_exists`, is very large. This is no surprise, since this intermediate result is indeed the BDD $\mathcal{E}(s, s')$, which we were avoiding by representing the partition using $\mathcal{P}(s', b)$.

The solution in [41] was to avoid computing $\mathcal{E}$ by computing the signatures and the refinement only with respect to one block at a time, which also enables several optimisations in [40].

```
1  def inert(T, P^s, P^s'):
2      if T = 0 : return 0
3      if result ← cache[(T, P^s, P^s')] : return result
       # interpret s'_i in P^s as s_i
4      v = topVar(T, P^s, P^s')
5      if v equals s_i for some i :
           # match s_i in T with s'_i in P^s
6          do in parallel:
7              low ← inert(T_{s_i=0}, P^s_{s'_i=0}, P^s')
8              high ← inert(T_{s_i=1}, P^s_{s'_i=1}, P^s')
9          result ← lookupBDDnode(s_i, low, high)
10     elif v equals s'_i for some i :
           # match s'_i in T with s'_i in P^s'
11         do in parallel:
12             low ← inert(T_{s'_i=0}, P^s, P^s'_{s'_i=0})
13             high ← inert(T_{s'_i=1}, P^s, P^s'_{s'_i=1})
14         result ← lookupBDDnode(s'_i, low, high)
15     else:
           # match the blocks P^s and P^s'
16         if P^s ≠ P^s' : result ← 0
17         else: result ← T
18     cache[(T, P^s, P^s'] ← result
19     return result
```

Algorithm 3 Computes the inert transitions of a transition relation $\mathcal{T}$ according to the block assignments to current states ($\mathcal{P}^s$) and next states ($\mathcal{P}^{s'}$).

We present an alternative solution, which computes $\rightarrow \cap \equiv$ directly using a custom BDD algorithm. The `inert` algorithm takes parameters $\mathcal{T}(s, s')$ ($\mathcal{T}$ may contain other variables ordered after s, s') and two copies of $\mathcal{P}(s', b)$: $\mathcal{P}^s$ and $\mathcal{P}^{s'}$. The algorithm matches $\mathcal{T}$ and $\mathcal{P}^s$ on valuations of variables s, and $\mathcal{T}$ and $\mathcal{P}^{s'}$ on valuations of variables s'. See Algorithm 3, and also Fig. 2 for a schematic overview. When in the recursive call all valuations to s and s' have been matched, with $S_s, S_{s'} \subseteq S$ the sets of states represented by these valuations, $\mathcal{T}$ is the set of actions that label the transitions between states in S_s and $S_{s'}$, $\mathcal{P}^s$ is the block that contains all S_s, and $\mathcal{P}^{s'}$ is the block that contains all $S_{s'}$. Then, if $\mathcal{P}^s \neq \mathcal{P}^{s'}$, the transitions are not inert and `inert` returns `False`, removing the transition from $\mathcal{T}$. Otherwise, $\mathcal{T}$ (which may still contain other variables ordered after s, s', such as action labels) is returned.

5 Quotient computation

Computing the partition of the maximal bisimulation is only the first part of the minimisation process. We must also apply the partition to the original system, such that the blocks of the partition become the states of the new transition system. A straightforward conversion procedure encodes the new states using the block numbers assigned during partition refinement.

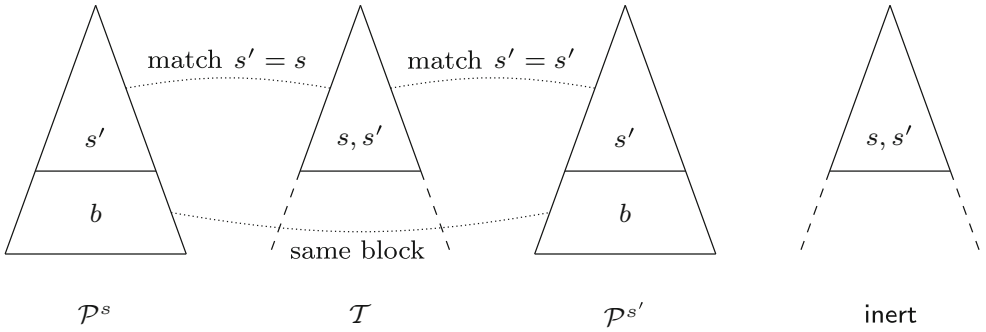

Fig. 2 Schematic overview of the BDDs in the `inert` algorithm

Just like partition refinement, the quotient can be computed with a sequence of standard BDD operations. We describe how the SIGREF tool by Wimmer et al. [41] implements this computation. Furthermore, we develop specialised algorithms which significantly speedup quotient computation for the interactive transition relation (Sect. 5.1) and for the Markovian transition relation (Sect. 5.2). Finally, we investigate a different encoding that does not use the assigned block numbers for the new system, but picks an arbitrary state from each block as a representative (Sect. 5.3).

5.1 Computing the new interactive transition relation

For LTSs and IMCs, the new interactive transition relation is computed using the original transition relation and the partition. We first describe how this relation is computed using standard BDD operations in the SIGREF tool [41]. We then present a new algorithm that performs all steps in one operation.

The SIGREF tool implements two methods to compute the new interactive transition relation. The first consists of the following steps:

1. Merge target states to the new encoding (in b).

$$\mathcal{T}(s, b, a) \;:=\; \exists s' \colon \mathcal{T}(s, s', a) \wedge \mathcal{P}(s', b)$$

2. Rename b variables to s' variables.

$$\mathcal{T}(s, s', a) \;:=\; \mathcal{T}(s, b, a)[b \leftarrow s']$$

3. Merge source states to the new encoding (in b).

$$\mathcal{T}(s', b, a) \;:=\; \exists s \colon \mathcal{T}(s, s', a) \wedge \mathcal{P}'(s, b)$$

4. Rename b variables to s variables.

$$\mathcal{T}(s, s', a) \;:=\; \mathcal{T}(s', b, a)[b \leftarrow s]$$

5. Remove τ-loops (only for branching bisimulation).

$$\mathcal{T}(s, s', a) \;:=\; \mathcal{T}(s, s', a) \wedge \neg(s = s' \wedge a = \tau)$$

Encoding and merging states (steps 1 and 3) are carried out using the BDD operation `and_exists` on the transition relation and the partition, where the existential quantification causes the transitions to states in the same block and from states in the same block to be combined like a set union. It is straightforward to see that the result is correct, as long as τ-loops are removed for branching bisimulation. For strong bisimulation, all states in a block have the same transitions, so existential quantification has no effect. For branching bisimulation, all states in a block can reach transitions via inert τ-steps, so combining the transitions with existential quantification is necessary to compute the correct result.

Step 1 requires the partition defined on s' and b variables, whereas step 3 requires the partition defined on s and b variables, in order to perform `and_exists`. Therefore, one additional rename operation is required to obtain a duplicate of the partition defined on the other variables. The algorithm to compute the quotient is then as follows:

```
1 def quotient(T(s, s', a), P(s', b)):
2     T(s, b, a) ← and_exists(T, P, s')
3     T(s, s', a) ← rename(T, [b ← s'])
4     P'(s, b) ← rename(P, [s' ← s])
5     T(s', b, a) ← and_exists(T, P', s)
6     T(s, s', a) ← rename(T, [b ← s])
      # for branching bisimulation:
7     T ← and(T, ¬(s = s' ∧ a = τ))
8     return T
```

Steps 1–5 coincide with lines 2–7 in the above algorithm. The BDD for $s = s' \wedge a = \tau$ (line 7) is trivial and can be computed just before line 7.

The SIGREF tool also implements a more optimised version, by introducing b' variables that are interleaved with the b variables, similar to how s and s' variables are interleaved.

1. Merge target states to the new encoding (in b').

$$\mathcal{T}(s, b', a) := \exists s' \colon \mathcal{T}(s, s', a) \wedge \mathcal{P}'(s', b')$$

2. Merge source states to the new encoding (in b).

$$\mathcal{T}(b, b', a) := \exists s \colon \mathcal{T}(s, a, b') \wedge \mathcal{P}''(s, b)$$

3. Rename b and b' variables to s and s' variables.

$$\mathcal{T}(s, s', a) := \mathcal{T}(a, b, b')[b \leftarrow s, b' \leftarrow s']$$

4. Remove τ-loops (only for branching bisimulation).

$$\mathcal{T}(s, s', a) := \mathcal{T}(s, s', a) \wedge \neg(s = s' \wedge a = \tau)$$

Since we use s' and b variables for $\mathcal{P}$, two rename operations would be required to compute $\mathcal{P}'(s', b')$ and $\mathcal{P}''(s, b)$. Instead, we perform this version as follows:

1. Merge target states to the new encoding (in b).

$$\mathcal{T}(s, b, a) := \exists s' \colon \mathcal{T}(s, s', a) \wedge \mathcal{P}(s', b)$$

2. Rename s and b variables to s' and b' variables.

$$\mathcal{T}(s', b', a) := \mathcal{T}(s, b, a)[s \leftarrow s', b \leftarrow b']$$

3. Merge source states to the new encoding (in b).

$$\mathcal{T}(b, b', a) := \exists s \colon \mathcal{T}(s', b', a) \wedge \mathcal{P}(s', b)$$

4. Rename b and b' variables to s and s' variables.

$$\mathcal{T}(s, s', a) := \mathcal{T}(b, b', a)[b \leftarrow s, b' \leftarrow s']$$

5. Remove τ-loops (only for branching bisimulation).

$$\mathcal{T}(s, s', a) := \mathcal{T}(s, s', a) \wedge \neg(s = s' \wedge a = \tau)$$

This procedure avoids creating a copy of $\mathcal{P}$ by renaming. The implementation is then as follows:

```
1 def quotient(T(s, s', a), P(s', b)):
2     T(s, b, a) ← and_exists(T, P, s')
3     T(s', b', a) ← rename(T, [s ← s', b ← b'])
4     T(b, b', a) ← and_exists(T, P, s)
5     T(s, s', a) ← rename(T, [b ← s, b' ← s'])
      # for branching bisimulation:
6     T ← and(T, ¬(s = s' ∧ a = τ))
7     return T
```

These algorithms still compute intermediate results that could be avoided by combining several steps into one operation. For example, every rename operation essentially creates a duplicate of the original BDD, when most BDD nodes are affected by the renaming. Using a custom operation can mitigate this. Similar to the `inert` algorithm discussed in Sect. 4.4, we implement the algorithm `quotient` that combines all steps of the above two algorithms. See Fig. 3 and Algorithm 4. Note the similarities with Fig. 2 and Algorithm 3.

```
1  def quotient(T, P^s, P^s'):
2      if T = 0 : return 0
3      if result ← cache[(T, P^s, P^s')] : return result
       # interpret s'_i in P^s as s_i
4      v = topVar(T, P^s, P^s')
5      if v equals s_i for some i :
           # match s_i in T with s'_i in P^s
6          low ← quotient(T_{s_i=0}, P^s_{s'_i=0}, P^s')
7          high ← quotient(T_{s_i=1}, P^s_{s'_i=1}, P^s')
8          result ← or(low, high)
9      elif v equals s'_i for some i :
           # match s'_i in T with s'_i in P^s'
10         low ← quotient(T_{s'_i=0}, P^s, P^s'_{s'_i=0})
11         high ← quotient(T_{s'_i=1}, P^s, P^s'_{s'_i=1})
12         result ← or(low, high)
13     else:
           # remove inert τ-loops (branching
             only)
14         if P^s = P^s' : T ← T ∧ ¬τ
           # convert blocks P^s and P^s'
15         result ← makecube(P^s,P^s',T))
16     cache[(T, P^s, P^s'] ← result
17     return result

18 def makecube(B^s, B^s', A, V = s ∪ s'):
19     if B^s = 0 ∨ B^s' = 0 : return 0
20     if V = ∅ : return A
21     v, V ← var(V), next(V)
22     if v equals s_i for some i :
23         low ← makecube(low(B^s),B^s',A,V)
24         high ← makecube(high(B^s),B^s',A,V)
25         return lookupBDDnode(v, low, high)
26     else:
27         low ← makecube(B^s,low(B^s'),A,V)
28         high ← makecube(B^s,high(B^s'),A,V)
29         return lookupBDDnode(v, low, high)
```

Algorithm 4 Computes the quotient of a transition relation $\mathcal{T}$ according to the block assignments to current states ($\mathcal{P}^s$) and next states ($\mathcal{P}^{s'}$).

Like the `inert` operation, we evaluate and match the transition relation with two copies of the partition (lines 1–12) and obtain the source block, the target block, and the set of actions at line 14–15. If we perform branching bisimulation and the source and target blocks are identical, we remove the τ transition from the obtained set of actions (line 14). As the two BDDs for the blocks are simple cubes that encode exactly one block by assigning a value to each b variable, and $\mathcal{T}$ is the set of actions A, it is very straightforward to compute

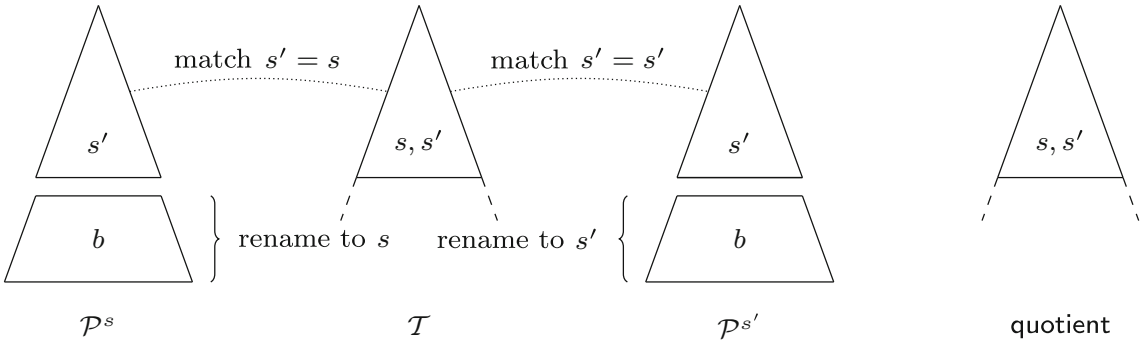

Fig. 3 Schematic overview of the BDDs in the `quotient` algorithm for interactive transition relations

the BDD representing the triple (s, s', A) using the recursive function `makecube` (line 15), which we included for completeness in Algorithm 4 at lines 18–29. Then, we combine all tuples computed at line 15 with `or` (lines 8 and 12), which has the same effect as existential quantification in the original algorithm.

5.2 Computing the new Markovian transition relation

For CTMCs and IMCs, the new Markovian transition relation must be computed. We first describe how this relation is computed using standard BDD operations in the SIGREF tool [41]. We then present a new algorithm that combines several steps of the computation.

The SIGREF tool uses the following method to compute the new Markovian transition relation:

1. Merge target states to the new encoding (in b).

   $$\mathcal{R}(s, b) := \exists_{\texttt{sum}} s' : \mathcal{R}(s, s') \wedge \mathcal{P}(s', b)$$

2. Rename b variables to s' variables.

   $$\mathcal{R}(s, s') := \mathcal{R}(s, b)[b \leftarrow s']$$

3. Merge source states to the new encoding (in b).

   $$\mathcal{R}(s', b) := \exists_{\texttt{max}} s : \mathcal{R}(s, s') \wedge \mathcal{P}'(s, b)$$

4. Rename b variables to s variables.

   $$\mathcal{R}(s, s') := \mathcal{R}(s', b)[b \leftarrow s]$$

First, the target states are converted to the new encoding using `and_exists_sum`, as transition rates to different states in the same block are added to obtain $\mathcal{R}(s, b)$. The variables b are renamed to s' to obtain $\mathcal{R}(s, s')$. The source states are then converted to the new encoding using `and_exists_max`, as we take the maximum, as discussed in Sect. 3, to obtain $\mathcal{R}(s', b)$. Finally, the variables b are renamed to s to obtain the result $\mathcal{R}(s, s')$.

The algorithm to compute the quotient is then as follows:

```
1 def quotient(R(s, s'), P(s', b)):
2     R(s, b) ← and_exists_sum(R, P, s')
3     R(s, s') ← rename(R, [b ← s'])
4     P'(s, b) ← rename(P, [s' ← s])
5     R(s', b) ← and_exists_max(R, P, s)
6     R(s, s') ← rename(R, [b ← s])
7     return R
```

We also implemented a custom `quotient` operation for the Markovian transition relation. However, not all steps can be combined like with interaction transition relation, since adding rates from states to blocks must be done before the source states are merged. Thus, we can only combine steps 2–4. The `quotient` operation for the Markovian transition relation is similar to the implementation of `and_exists_max` in Sylvan, modified to perform the `rename` operations on the fly and we omit it due to space limitations.

5.3 Alternative encoding for new states

The standard encoding of the states in the new transition system uses the block numbers assigned during partition refinement. This can have a significant disadvantage. Symbolic models are powerful as they can represent large state spaces efficiently by exploiting structural properties of the transition system, like symmetries and independent variables. Such properties are lost when using the block numbers of the partition.

We propose an alternative encoding "pick-one-state" that picks one state from each block to represent all states in the block. Each path in $\mathcal{P}$ to the sub-BDD that represents a block (on b variables) encodes states in that block, such that state variables encountered along the path are `True` if the high edge was followed and `False` if the low

```
1  def pick(P, path):
2      if P = 0 : return
3      if cache[P] : return
4      cache[P] ← ⊤
5      v = var(P)
6      if v is a block variable :
7          B ← decodeBlock(P)
8          if picked[B] = ⊥ :
9              picked[B] ← pick_one_state(path)
10     else:
11         do in parallel:
12             pick(P_{v=0}, path + ¬v)
13             pick(P_{v=1}, path + v)
```

Algorithm 5 Algorithm `pick` to obtain one state for each block in the partition.

edge was followed. We use this information to compute exactly one state (encoded using b variables, with missing state variables set to `False`) that represents the block and store this state in an array. Since we are simply interested in obtaining one state that represents each block, we only need to visit each node in the BDD $\mathcal{P}$ once, so we use the operation cache to denote whether we have visited the node. See Algorithm 5. This algorithm `pick` fills an array `picked` with a single state for each block, obtained from the path as described above using a helper function `pick_one_state`.

After obtaining a single state for each block, we can use an algorithm similar to `refine` (Sect. 4.3) to replace each block in $\mathcal{P}$ by the selected state (encoded using b variables). Then, the same algorithms as in Sects. 5.1 and 5.2 compute the new transition system using the proposed encoding.

6 Tool support

We implemented multi-core symbolic signature-based bisimulation minimisation in a tool called SIGREFMC. The tool supports LTSs, CTMCs, and IMCs delivered in two input formats, the XML format used by the original SIGREF tool and the BDD format that the tool LTSMIN [28] generates for various model checking languages. SIGREFMC supports both the floating-point and the rational representation of rates in continuous-time transitions.

One of the design goals of this tool is to encourage researchers to extend it for their own file formats and notions of bisimulation, and to integrate it in other toolsets. Therefore, SIGREFMC is freely available online[1] and licensed with the permissive Apache 2.0 license. Documentation is available and instructions for extending the tool for different input/output formats and types of bisimulation are included.

6.1 Support for LTSMIN

SIGREFMC supports models are generated by the model checking toolset LTSMIN. LTSMIN provides a language-independent Partitioned Next-State Interface (PINS), which connects various input languages to model checking algorithms [6,28,31]. In PINS, the states of a system are represented by vectors of N integer values. Furthermore, transitions are distinguished in K disjunctive "transition groups", i.e., each transition in the system belongs to one of these transition groups. The transition relation of each transition group usually only depends on a subset of the entire state vector called the "short vector", further distinguished by the variables that are "read" and the variables that are "written" [31]. This enables the efficient encoding of transitions that only affect some integers of the state vector. Exploiting this information lets the PINS interface work in a quasi-symbolic way, as a single pair of short vectors can represent many transition relations on the full state vector.

Initially, LTSMIN does not have knowledge of the transitions in each transition group, and only the initial state is known. The transition system is explored by learning new transitions via the PINS interface, which are then added to the transition relation. Various input languages connect to LTSMIN via the PINS interface by implementing a `next-state` function, which produces all target states (as write vectors) reachable from a given source state (as read vector). Using the LTSMIN toolset, we can convert process algebra specifications in the language MCRL2 [13] to the BDD file format that SIGREFMC supports. We can then minimise the obtained LTS using the techniques described in this paper and obtain the result, either as a symbolic LTS or as a simple explicit-state enumeration of transitions between states.

7 Experimental evaluation

This section reports on the experimental evaluation of the techniques proposed in this paper. We study the improvements to signature refinement in Sect. 7.1, the improvements to quotient computation in Sect. 7.2, the effect of ordering block variables after or before action variables in Sect. 7.3, and finally the performance of the presented tool SIGREFMC on process algebra benchmarks produced with LTSMIN in Sect. 7.4. We also refer to the full experimental data that are available online[2] and can be reproduced.

When comparing SIGREFMC to other tools, we restrict ourselves to the symbolic bisimulation minimisation tool SIGREF by Wimmer et al., as [41] already compares SIGREF to

[1] https://github.com/utwente-fmt/sigrefmc.

[2] https://github.com/utwente-fmt/sigrefmc-sttt16.

Table 1 Computation time in seconds for partition refinement on the benchmarks, comparing SIGREF with SIGREFMC

Model	States	Blocks	Time			Speedups		
			T_w	T_1	T_{48}	Seq.	Par.	Total
LTS models (strong)								
kanban03	1,024,240	85,356	92.16	10.09	0.88	9.14×	11.52×	105.29×
kanban04	16,020,316	778,485	1410.66	148.15	11.37	9.52×	13.03×	124.06×
kanban05	16,772,032	5,033,631	–	1284.86	73.57	–	17.47×	–
kanban06	264,515,056	25,293,849	–	–	2584.23	–	–	–
LTS models (branching)								
kanban04	16,020,316	2785	8.47	0.52	0.24	16.39×	2.11×	34.60×
kanban05	16,772,032	7366	34.11	1.48	0.43	22.98×	3.47×	79.81×
kanban06	264,515,056	17,010	118.19	3.87	0.83	30.55×	4.65×	142.20×
kanban07	268,430,272	35,456	387.16	8.83	1.66	43.86×	5.31×	232.71×
kanban08	4,224,876,912	68,217	1091.67	17.91	2.98	60.96×	6.02×	366.72×
kanban09	4,293,193,072	123,070	3186.48	34.23	5.51	93.10×	6.21×	578.59×
CTMC models								
cycling-4	431,101	282,943	220.23	26.72	2.60	8.24×	10.29×	84.84×
cycling-5	2,326,666	1,424,914	1249.23	170.28	19.42	7.34×	8.77×	64.34×
fgf	80,616	38,639	71.62	8.86	0.88	8.08×	10.04×	81.20×
p2p-5-6	2^{30}	336	750.29	26.96	2.99	27.83×	9.03×	251.24×
p2p-6-5	2^{30}	266	248.17	9.49	1.21	26.15×	7.82×	204.47×
p2p-7-5	2^{35}	336	2280.76	24.01	2.97	94.99×	8.08×	767.12×
polling-16	1,572,864	98,304	792.82	118.50	10.18	6.69×	11.64×	77.85×
polling-17	3,342,336	196,608	1739.01	303.65	22.58	5.73×	13.45×	77.03×
polling-18	7,077,888	393,216	–	705.22	49.81	–	14.16×	–
robot-020	31,160	30,780	28.15	3.21	0.60	8.78×	5.36×	47.04×
robot-025	61,200	60,600	78.48	6.78	0.95	11.58×	7.11×	82.39×
robot-030	106,140	105,270	174.30	12.26	1.47	14.21×	8.33×	118.44×
IMC models (strong)								
ftwc01	2048	1133	1.26	1.14	0.2	1.11×	5.76×	6.38×
ftwc02	32,768	16,797	154.55	102.07	15.85	1.51×	6.44×	9.75×
IMC models (branching)								
ftwc01	2048	430	1.12	0.77	0.13	1.45×	6.07×	8.83×
ftwc02	32,786	3886	152.9	50.39	4.89	3.03×	10.3×	31.26×

Each data point is an average of at least 15 runs. The timeout was 3600 s

other explicit-state and symbolic bisimulation minimisation tools.

7.1 Signature refinement

7.1.1 Design

To study the improvements to signature refinement that we present in this paper, we compared our results (using the skip list variant of `refine`) to SIGREF 1.5 [40] for LTS and IMC models, and to a version of SIGREF used in [38] for CTMC models. For the CTMC models, we used SIGREF with rational numbers provided by the GMP library and SIGREFMC with rational number support by Sylvan. For the IMC models, version 1.5 of SIGREF does not support the GMP library and the version used in [38] does not support IMCs. We used SIGREFMC with floating points for a fairer comparison, but the tools give a slightly different number of blocks, due to the use of floating points.

We restrict ourselves to the models presented in [38,41] and an IMC model that is part of the distribution of SIGREF. These models have been generated from PRISM benchmarks using a custom version of the PRISM toolset [30]. We refer to the literature for a description of these models.

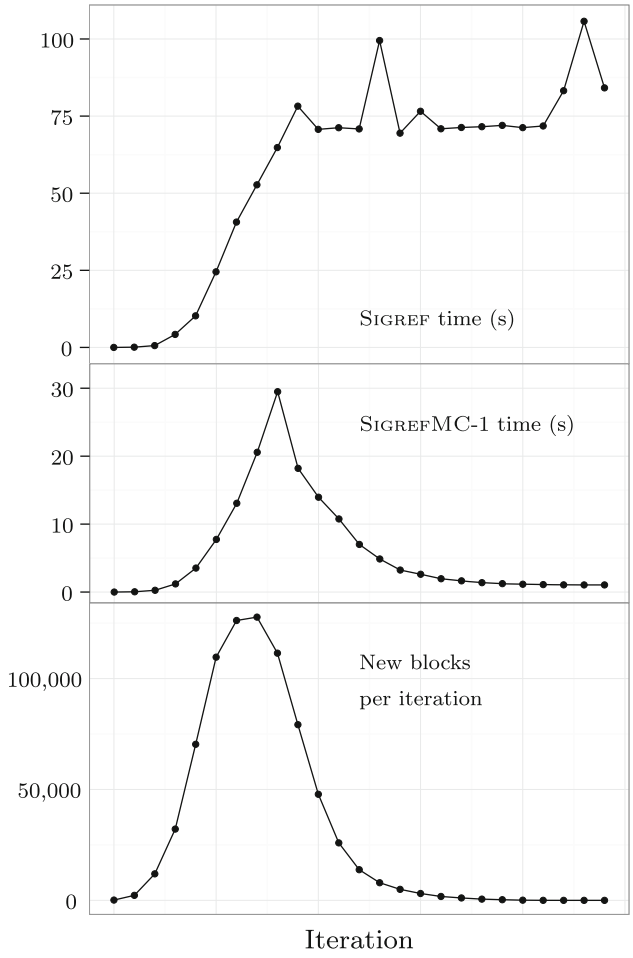

Fig. 4 Time per iteration for SIGREF and SIGREFMC (1 worker), and the number of new blocks per iteration for strong bisimulation of the kanban04 LTS model

We perform experiments on the three tools using a 48-core machine, containing 4 AMD Opteron™ 6168 processors with 12 cores each. We measure the runtimes for the partition refinement algorithm (excluding file-I/O) using SIGREF, SIGREFMC with only 1 worker, and SIGREFMC with 48 workers.

Apart from the new `refine` and `inert` algorithms presented in the current paper, there are several other differences. The first is that the original SIGREF uses the CUDD implementation of BDDs, while SIGREFMC uses Sylvan, along with some extra BDD algorithms that avoid explicitly computing variable renaming of some BDDs. The second is that SIGREF has several optimisations [40] that are not available in SIGREFMC.

7.1.2 Results

See Table 1 for the results of these experiments. These results were obtained by repeating each benchmark at least 15 times and taking the average. The timeout was set to 3600 s. The column "States" shows the number of states before bisimulation minimisation and "Blocks" the number of equivalence classes after bisimulation minimisation. We show the wall clock time using SIGREF (T_w), using SIGREFMC with 1 worker (T_1) and using SIGREFMC with 48 workers (T_{48}). We compute the sequential speedup T_w/T_1, the parallel speedup T_1/T_{48}, and the total speedup T_w/T_{48}.

Note that we obtained these results using the variable ordering $s, s' < a < b$; the other experiments are computed using the variable ordering $s, s' < b < a$, as discussed below and in Sect. 4.2.

Due to space constraints, we do not include all results, but restrict ourselves to larger models. We refer to the full experimental data that is available online. In the full set of results, excluding executions that take less than 1 s, SIGREFMC is always faster sequentially and always benefits from parallelism.

The results show a clear advantage for larger models. One interesting result is for the p2p-7-5 model. This model is ideal for symbolic bisimulation with a large number of states (2^{35}) and very few blocks after minimisation (336). For this model, our tool is 95× faster sequentially and has a parallel speedup of 8×, resulting in a total speedup of 767×. The best parallel speedup of 17× was obtained for the kanban05 model.

In almost all experiments, the signature computation dominates with 70–99% of the execution time sequentially. We observe that the refinement step sometimes benefits more from parallelism than signature computation, with speedups up to 29.9×. We also find that reusing block numbers for stable blocks causes a major reduction in computation time towards the end of the procedure. The kanban LTS models and the larger polling CTMC models are an excellent case study to demonstrate this. See Fig. 4. There is a clear correlation between the number of new blocks per iteration and the time per iteration for SIGREFMC, while the time per iteration for SIGREF seems to correlate with the number of blocks.

7.2 Quotient computation

7.2.1 Design

To study the different methods for quotient computation, we implemented the methods described in Sects. 5.1 and 5.2:

- `block-s`: block encoding using standard operations
- `block`: block encoding using specialised operations
- `pick`: pick-one-state encoding, specialised operations

We computed the partition in SIGREFMC using rational numbers for the Markovian transitions and with the variable

Table 2 Computation time in seconds for different implementations of quotient computation

	`block-s`			`block`			`pick`		
	T_1	T_{48}	Sp.	T_1	T_{48}	Sp.	T_1	T_{48}	Sp.
LTS model (strong)									
kanban03	24.64	1.5	16.42×	9.48	0.48	19.85×	6.72	0.35	19.08×
kanban04	370.16	21.25	17.42×	129.19	7.84	16.47×	106.22	5.38	19.73×
kanban05	–	175.92	–	1114.06	55.26	20.16×	740.53	33.80	21.91×
LTS model (branching)									
kanban04	1.08	0.12	8.91×	0.20	0.03	6.67×	0.16	0.04	3.65×
kanban05	3.48	0.33	10.71×	0.68	0.09	7.60×	0.51	0.10	5.05×
kanban06	11.44	1.10	10.38×	1.90	0.27	6.95×	1.42	0.30	4.78×
kanban07	29.94	3.02	9.93×	5.38	0.77	7.00×	3.17	0.64	4.93×
kanban08	110.47	8.34	13.24×	11.52	1.52	7.56×	7.01	1.29	5.44×
kanban09	200.44	18.77	10.68×	27.05	3.83	7.06×	14.21	2.74	5.19×
CTMC model									
cycling-4	170.2	9.51	17.91×	40.22	3.05	13.21×	59.51	3.32	17.90×
cycling-5	1039.17	55.52	18.72×	231.25	14.01	16.50×	294.15	13.48	21.83×
fgf	17.77	1.64	10.83×	6.12	0.61	9.99×	7.42	0.73	10.20×
kanban-3	19.32	1.5	12.87×	6.4	0.58	11.07×	7.04	0.49	14.26×
kanban-4	285.52	14.72	19.40×	81.57	4.67	17.48×	104.65	5.08	20.60×
p2p-5-6	22.1	2.34	9.45×	9.66	1.12	8.63×	10.25	1.41	7.29×
p2p-6-5	7.45	0.91	8.17×	3.41	0.45	7.64×	3.67	0.55	6.71×
p2p-7-5	17.55	2.02	8.71×	8.84	1.05	8.39×	9.26	1.19	7.79×
polling-16	176.47	8.74	20.20×	95.33	4.83	19.76×	66.25	4.49	14.75×
polling-17	416.17	20.65	20.16×	223.11	11.51	19.39×	161.74	10.02	16.14×
polling-18	1063.13	53.38	19.92×	542.02	26.43	20.51×	359.49	21.68	16.58×
robot-020	3.47	0.27	12.68×	1.72	0.16	10.83×	1.55	0.12	12.57×
robot-025	6.97	0.54	13.00×	3.39	0.32	10.66×	2.91	0.25	11.83×
robot-030	12.36	1.03	12.04×	5.84	0.53	10.98×	4.81	0.41	11.78×
IMC model (strong)									
ftwc01	1.62	0.16	10.06×	1.69	0.14	12.22×	0.96	0.08	11.98×
ftwc02	208.89	20.78	10.05×	370.16	36.65	10.10×	301.88	15.34	19.68×
IMC model (branching)									
ftwc01	0.36	0.05	6.99×	0.3	0.03	9.00×	0.19	0.03	6.83×
ftwc02	17.13	1.72	9.98×	15.73	1.45	10.86×	5.24	0.49	10.77×

Each data point is an average of at least 12 runs. The timeout was 1200 s to compute the partition and the quotient

ordering $s, s' < b < a$ for the interactive transitions. We used the same 48-core machine as for the experiments in Sect. 7.1. We measure the time for quotient computation with 1 worker and with 48 workers. Our experimental setup performed all benchmarks in random order and repeated the experiments ad infinitum. When we halted the script, every benchmark was performed at least 12×. The timeout was set to 1200 s, including time to compute the partition.

7.2.2 Results

See Table 2 for the results of these experiments. The results show that the `block` implementation is faster than the `block-s` implementation, except for the ftwc02 model. For CTMC models, using specialised operations results in a speedup of 2–3×. For LTS models, using specialised operations results in a speedup of 5–9×. The pick-one-state encoding shows mixed results for computation time, as it can be slower or faster than block encoding. Furthermore, we obtain a parallel speedup of up to 20.5× for the block encoding and 21.9× with the pick-one-state encoding, with 48 workers.

See Table 3 for the sizes of the computed transition relations using block encoding and using pick-one-state encoding, in number of BDD nodes. In many cases, pick-one-state encoding is superior, with up to 5162× smaller BDDs

Table 3 Number of BDD nodes for the transition relation after quotient computation, for the block number encoding and the pick-one-state encoding

	block	pick	factor
LTS (strong)			
kanban03	710,359	6137	115.75×
kanban04	6,553,843	14,599	448.92×
kanban05	43,901,839	27,600	1590.65×
LTS (branching)			
kanban04	17,510	1081	16.20×
kanban05	47,920	1259	38.06×
kanban06	110,069	1944	56.62×
kanban07	233,902	1999	117.01×
kanban08	442,890	2838	156.06×
kanban09	800,649	3388	236.32×
IMC (strong)			
ftwc01	47,859	660	72.51×
ftwc02	5,669,528	1208	4693.32×
IMC (branching)			
ftwc01	2137	285	7.50×
ftwc02	49,093	413	118.87×
ftwc03	1,236,052	541	2284.75×
CTMC			
cycling-4	1,869,641	185,824	10.06×
cycling-5	8,960,365	430,936	20.79×
fgf	422,954	38,452	11.00×
kanban-3	354,774	2473	143.46×
kanban-4	3,032,327	4899	618.97×
p2p-5-6	1513	2635	0.57×
p2p-6-5	1039	2151	0.48×
p2p-7-5	1428	3057	0.47×
polling-16	715,145	494	1447.66×
polling-17	1,442,013	529	2725.92×
polling-18	2,901,462	562	5162.74×
robot-020	148,385	3790	39.15×
robot-025	260,514	4785	54.44×
robot-030	411,624	5512	74.68×

for the polling models. For the p2p models, block encoding is superior, likely due to the small number of blocks after bisimulation minimisation.

7.3 Variable ordering

7.3.1 Design

As discussed in Sect. 4.2, we can choose to order block variables b before or after action variables a in the variable ordering of the BDDs. To compare the ordering $s, s' < a < b$ and $s, s' < b < a$, we compare signature refinement and quotient computation for the kanban LTS models.

We expect that in general ordering b before a is the best choice. If we have a variables before b variables, then it is guaranteed that all BDD nodes on a variables are recreated when we compute signatures for partition refinement and when we compute the quotient, whereas they may be reused if a variables are last in the ordering.

7.3.2 Results

See Table 4 for the results of this experiment. All data points are computed with at least 5 runs. We computed the quotient using the pick-one-state algorithm. We see that in most cases the ordering with b before a is superior. We observe a stronger effect for partition refinement than for quotient computation. The surprising exception is quotient computation of the kanban04 model with strong bisimulation, where the ordering with a before b is slightly better, although the total time still favours ordering b before a.

7.4 Process algebra experiments

7.4.1 Design

As described in Sect. 6.1, we extended SIGREFMC with support for BDDs produced by the model checking toolset LTSMIN from process algebra models specified in the MCRL2 specification language.

We first took a number of communication protocols from the MCRL2 example directory, in particular the bounded retransmission protocol (BRP) and the Sliding Window Protocol (SWP). We made them parametric in the number of data elements, number of retries, window size, etc. We also include a number of distributed algorithms. We ported the probabilistic leader election protocols [3], based on Dolev–Klawe–Rodeh and Franklin, from μCRL to MCRL2. We also included Hesselink's hardware register [27]. Finally, we also included an industrial case study: Workload Management System of the computation grid at the Large Hadron Collider LHC (CERN), specified in [36].

This leads to the following specifications:

- SWP_m_n: the Sliding Window Protocol [1] on m data items, with window size n. This specifies a one-directional version of the sliding window protocol. n subsequent data items can be sent and acknowledged in arbitrary order. This requires sequence numbers modulo $2n$. Its external behaviour is equivalent to a $2n$-place buffer.
- BRP_m_ℓ_n: the bounded retransmission protocol [24] on m data items, sending a list of length ℓ and with n

Table 4 Computation time in seconds on the LTS benchmarks, with the variable orders $s, s' < a < b$ and $s, s' < b < a$, for both partition refinement and quotient computation, with 1 worker and 48 workers

Model (bisimulation)	Partition, 1 worker		Partition, 48 workers		Quotient, 1 worker		Quotient, 48 workers	
	$a < b$	$b < a$	$a < b$	$b < a$	$a < b$	$b < a$	$a < b$	$b < a$
kanban03 (strong)	8.69	6.86	1.10	1.01	6.83	6.72	0.36	0.35
kanban04 (strong)	127.54	102.11	13.86	11.66	98.12	106.22	4.25	5.38
kanban05 (strong)	1211.20	1076.17	99.63	95.09	854.62	740.53	34.17	33.80
kanban04 (branching)	0.40	0.38	0.22	0.23	0.16	0.16	0.04	0.04
kanban05 (branching)	1.12	1.05	0.43	0.39	0.51	0.51	0.11	0.10
kanban06 (branching)	2.88	2.65	0.92	0.89	1.42	1.42	0.30	0.30
kanban07 (branching)	6.46	5.95	2.06	2.21	3.18	3.17	0.65	0.64
kanban08 (branching)	13.09	11.95	4.27	3.60	7.04	7.01	1.33	1.29
kanban09 (branching)	24.37	22.24	7.28	6.99	14.47	14.21	3.01	2.74

retries. This protocol extends the ABP, but gives up after n retries. The status of the transmission is returned to both the sender and the receiver. The external behaviour is a bit complicated, since the sender cannot distinguish if the last data element or the last acknowledgement got lost.

- DKR_n: randomised variant [3] of Dolev–Klawe–Rodeh's [22] Leader Election Protocol on a uni-directional ring with n anonymous partners. Several rounds may be needed when partners choose the same identity. The protocol is based on hop counters and on an alternating bit to distinguish subsequent rounds. The external behaviour is equivalent to a single leader action.
- Franklin_n_m: randomised variant [3] of Franklin's Leader Election Protocol [23], but now on a bidirectional ring with n partners, using $m \leq n$ different identities. The external behaviour is again equivalent to a single leader action.
- Hesselink_n: Hesselink's handshake register [27], constructed from four safe registers and four Boolean atomic registers, modelled in mCRL2 by Groote, and used for experimentation in [34].
- WMS: this models the Workload Management System of the DIRAC (Distributed Infrastructure with Remote Agent Control) for the Large Hadron Collider experiments at CERN, as described in [36].

We used the following toolchain to generate input files for SigrefMC:

1. `mcrl22lps -Dfvn` from the mCRL2 toolset to generate LPS files from the specifications
2. `lps2lts-sym --vset=lddmc` from the LTSmin toolset to generate the transition systems in LDD format from the LPS files
3. `ldd2bdd` from the LTSmin toolset to convert the transition systems from LDDs to BDDs

To evaluate SigrefMC on these models, we performed the same experiments as in Sect. 7.2.

We measure the time for partition refinement and quotient computation with 1 worker and with 48 workers. Our experimental setup performed all benchmarks in random order and repeated the experiments ad infinitum. When we halted the script, every benchmark was performed at least 6×. The timeout was set to 1200 s for the entire program, i.e., partition refinement and quotient computation.

7.4.2 Results

The results are summarised in Tables 5 and 6. We do not include all results to conserve space; all results from the experiments are available online.

It is interesting to see that both strong and branching bisimulation result in huge reductions. We see clear benefit from parallel processing, with speedups of up to 24.7× for signature refinement and up to 24.5× for quotient computation (block encoding)

The pick-one-state encoding does not work so well here. Probably because the number of blocks is low; also the state vectors are relatively long. For a few models, the pick-one-state encoding works relatively well; these are models that have a high number of blocks.

8 Conclusions

Originally, we intended to investigate parallelism in symbolic bisimulation minimisation. To our surprise, we obtained a much higher sequential speedup using specialised BDD operations, as demonstrated by the results in Table 1 and Fig. 4.

Table 5 Results for the process algebra benchmarks generated with LTSMIN

Model	States	Blocks	Signature refinement			Quotient (`block-s`)			Quotient (`block`)		
			T_1	T_{48}	Sp.	T_1	T_{48}	Sp.	T_1	T_{48}	Sp.
LTS model (strong)											
brp-2-4-4	11,182	2976	3.92	0.35	11.23×	1.32	0.43	3.08×	0.57	0.04	13.26×
brp-3-4-4	40,592	10,326	13.50	0.92	14.75×	5.30	0.63	8.48×	2.45	0.14	17.53×
brp-4-4-4	109,422	27,106	38.91	2.23	17.43×	18.03	1.49	12.10×	9.84	0.52	18.93×
dkr-3	11,455	208	7.64	0.54	14.05×	3.64	0.38	9.57×	1.31	0.09	14.50×
dkr-4	909,593	3429	–	115.28	–	–	25.86	–	–	4.99	–
franklin-3-2	11,805	702	7.15	0.47	15.21×	4.71	0.42	11.12×	0.99	0.07	13.55×
franklin-3-3	41,401	883	24	1.24	19.40×	16.25	1.05	15.52×	3.13	0.19	16.46×
franklin-4-2	272,241	10,706	330.56	14.67	22.53×	204.68	9.65	21.21×	28.04	1.43	19.63×
franklin-4-3	5,269,441	17,738	–	441.56	–	–	115.02	–	–	13.87	–
hesselink-2	540,736	1018	3.49	0.34	10.30×	1.96	0.30	6.60×	0.43	0.07	5.94×
hesselink-3	13,834,800	2835	17.70	1.42	12.50×	16.16	1.57	10.27×	2.33	0.35	6.58×
hesselink-4	142,081,536	6036	51.41	3.56	14.44×	66.71	5.37	12.43×	7.01	1.21	5.78×
hesselink-5	883,738,000	11,005	179.85	12.61	14.26×	313.42	25.40	12.34×	22.32	3.64	6.14×
swp-2-4	2,589,056	69,555	267.46	11.33	23.60×	258.66	13.40	19.30×	30.78	1.39	22.21×
swp-3-2	52,380	4710	4.12	0.25	16.45×	4.98	0.39	12.71×	0.73	0.05	14.57×
swp-3-3	1,652,724	65,025	142.60	6.13	23.26×	188.10	9.60	19.60×	24.89	1.11	22.39×
swp-4-2	140,352	11,553	9.77	0.54	18.02×	13.18	0.98	13.40×	1.96	0.12	16.10×
swp-4-3	7,429,632	–	630.73	25.92	24.34×	–	47.05	–	111.69	4.55	24.56×
WMS	155,034,776	1	0.12	0.02	4.91×	0.11	0.20	0.56×	0.10	0.13	0.79×
LTS model (branching)											
brp-2-4-4	11,182	98	3.63	0.36	10.11×	0.28	0.10	2.67×	0.18	0.02	7.71×
brp-3-4-4	40,592	328	13.78	0.98	14.08×	0.28	0.10	2.67×	0.18	0.02	7.71×
brp-4-4-4	109,422	858	39.71	2.16	18.38×	4.04	0.48	8.47×	4.52	0.23	19.64×
dkr-3	11,455	2	4.46	0.33	13.39×	0.94	0.38	2.47×	0.63	0.05	11.79×
dkr-4	909,593	2	349.24	15.31	22.81×	45.30	10.60	4.27×	25.73	1.37	18.82×
franklin-3-2	11,805	2	3.62	0.29	12.58×	0.53	0.35	1.50×	0.28	0.04	6.64×
franklin-3-3	41,401	2	11.94	0.66	17.96×	1.80	0.47	3.88×	0.95	0.07	13.55×
franklin-4-2	272,241	2	50.97	2.40	21.28×	4.76	1.76	2.71×	2.19	0.18	12.28×
franklin-4-3	5,269,441	2	807.72	32.69	24.71×	67.70	22.37	3.03×	31.94	1.56	20.52×
hesselink-2	540,736	72	7.64	0.79	9.71×	0.73	0.15	4.80×	0.19	0.03	6.33×
hesselink-3	13,834,800	189	37.10	2.76	13.46×	5.88	0.66	8.86×	0.94	0.13	7.36×
hesselink-4	142,081,536	384	114.37	7.98	14.33×	26.66	2.05	12.97×	2.79	0.38	7.44×
hesselink-5	883,738,000	675	351.69	23.93	14.70×	102.95	7.38	13.95×	8.33	1.11	7.49×
swp-2-4	2,589,056	511	116.16	5.08	22.88×	20.58	1.33	15.53×	2.32	0.13	18.09×
swp-3-2	52,380	121	4.41	0.31	14.07×	0.67	0.09	7.76×	0.11	0.01	11.00×
swp-3-3	1,652,724	1093	135.99	6.21	21.88×	18.35	1.16	15.84×	2.34	0.12	19.51×
swp-4-2	140,352	341	8.13	0.46	17.85×	1.96	0.34	5.74×	0.28	0.03	11.13×
swp-4-3	7,429,632	5461	420.09	17.13	24.52×	99.64	5.42	18.38×	10.59	0.49	21.68×
WMS	155,034,776	1	0.36	0.22	1.66×	0.11	0.22	0.51×	0.10	0.11	0.93×

We compute the partition using SIGREFMC and the quotient using the `block-s` and the `block` algorithms and give the computation time in seconds

The specialised BDD operations offer a clear advantage sequentially and the integration with Sylvan results in decent parallel speedups. Our best result had a total speedup of 767×. By also using specialised BDD operations for quotient computation, we demonstrated performance improvements in 2–10× over using standard BDD operations.

The success of this approach suggests that for applications that involve decision diagrams, specialised operations

Table 6 Results for the process algebra benchmarks generated with LTSMIN

Model	States	Blocks	quotient (`pick`)			Number of nodes		
			T_1	T_{48}	Sp.	`block`	`pick`	Factor
LTS model (strong)								
brp-2-4-4	11,182	2976	0.68	0.05	13.60×	9383	10,390	0.9×
brp-3-4-4	40,592	10,326	2.85	0.16	17.42×	22,981	21,935	1.05×
brp-4-4-4	109,422	27,106	10.94	0.58	18.69×	43,414	48,777	0.89×
dkr-3	11,455	208	1.48	0.10	14.85×	1192	31,412	0.04×
dkr-4	909,593	3429	–	5.44	–	–	–	–
franklin-3-2	11,805	702	1.27	0.10	13.32×	3706	65,776	0.06×
franklin-3-3	41,401	883	3.80	0.23	16.27×	4840	101,813	0.05×
franklin-4-2	272,241	10,706	38.27	1.86	20.52×	58,428	799,831	0.07×
franklin-4-3	5,269,441	17,738	–	15.79	–	–	–	–
hesselink-2	540,736	1018	0.60	0.11	5.76×	5927	8368	0.71×
hesselink-3	13,834,800	2835	3.13	0.54	5.77×	12,575	16,965	0.74×
hesselink-4	142,081,536	6036	9.32	1.81	5.15×	20,648	25,722	0.8×
hesselink-5	883,738,000	11,005	28.71	4.92	5.83×	32,335	43,141	0.75×
swp-2-4	2,589,056	69,555	50.90	2.47	20.62×	485,607	154,904	3.13×
swp-3-2	52,380	4710	1.20	0.09	13.09×	40,718	23,401	1.74×
swp-3-3	1,652,724	65,025	–	–	–	435,339	–	–
swp-4-2	140,352	11,553	3.08	0.23	13.22×	93,494	40,475	2.31×
swp-4-3	7,429,632	264,708	164.38	6.69	24.56×	1,474,564	404,756	3.64×
WMS	155,034,776	1	0.11	0.10	1.05×	7	265	0.03×
LTS model (branching)								
brp-2-4-4	11,182	98	0.20	0.02	10×	804	4514	0.18×
brp-3-4-4	40,592	328	1.00	0.06	16.16×	2136	12,221	0.17×
brp-4-4-4	109,422	858	4.67	0.24	19.33×	4383	31,192	0.14×
dkr-3	11,455	2	0.68	0.06	11.81×	5	163	0.03×
dkr-4	909,593	2	26.40	1.43	18.5×	5	251	0.02×
franklin-3-2	11,805	2	0.29	0.03	8.8×	5	139	0.04×
franklin-3-3	41,401	2	1.00	0.07	14.26×	5	175	0.03×
franklin-4-2	272,241	2	2.27	0.18	12.7×	5	203	0.02×
franklin-4-3	5,269,441	2	33.04	1.59	20.8×	5	267	0.02×
hesselink-2	540,736	72	0.22	0.03	7.33×	653	3700	0.18×
hesselink-3	13,834,800	189	1.10	0.14	7.62×	1516	9191	0.16×
hesselink-4	142,081,536	384	3.36	0.42	7.94×	2329	14,253	0.16×
hesselink-5	883,738,000	675	9.94	1.22	8.14×	3749	24,943	0.15×
swp-2-4	2,589,056	511	2.80	0.14	19.99×	1821	4722	0.39×
swp-3-2	52,380	121	0.13	0.01	13×	555	2620	0.21×
swp-3-3	1,652,724	1093	2.75	0.14	20.3×	3994	10,461	0.38×
swp-4-2	140,352	341	0.33	0.02	14×	1588	4952	0.32×
swp-4-3	7,429,632	5461	12.28	0.56	21.87×	15,050	26,941	0.56×
WMS	155,034,776	1	0.11	0.11	0.97×	3	89	0.03×

We compute the partition using SIGREFMC and the quotient using the `block-s` and the `block` algorithms and give the computation time in seconds

that combine sequential steps can be a good method to obtain performance improvements in several orders of magnitude. Similarly, the additional performance improvement gained by the parallel framework from Sylvan is relatively low hanging fruit to improve the performance of symbolic algorithms with decision diagrams.

The pick-one-state encoding that we proposed in this paper is promising, especially for transition systems that

are still relatively large after bisimulation minimisation. The implementation discussed here just picked an arbitrary state; we expect that better heuristics may be developed in the future.

A limitation of this study is that we only measured the performance on the benchmarks that were used in [38,40] and on several benchmarks from the MCRL2 distribution.

Acknowledgements Open access funding provided by Johannes Kepler University Linz.

References

1. Badban, B., Fokkink, W., Groote, J.F., Pang, J., van de Pol, J.: Verification of a sliding window protocol in μCRL and PVS. Formal Asp. Comput. **17**(3), 342–388 (2005)
2. Bahar, R.I., Frohm, E.A., Gaona, C.M., Hachtel, G.D., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. ICCAD **1993**, 188–191 (1993)
3. Bakhshi, R., Fokkink, W., Pang, J., van de Pol, J.: Leader election in anonymous rings: Franklin goes probabilistic. In: Ausiello, G., Karhumäki, J., Mauri, G., Ong, C.L. (eds.) TCS'08, IFIP, vol. 273, pp. 57–72. Springer, Berlin (2008)
4. Blom, S., Haverkort, B.R., Kuntz, M., van de Pol, J.: Distributed Markovian bisimulation reduction aimed at CSL model checking. ENTCS **220**(2), 35–50 (2008)
5. Blom, S., Orzan, S.: Distributed branching bisimulation reduction of state spaces. ENTCS **89**(1), 99–113 (2003)
6. Blom, S., van de Pol, J., Weber, M.: LTSmin: distributed and symbolic reachability. In: CAV, LNCS, vol. 6174, pp. 354–359. Springer (2010)
7. Blumofe, R.D.: Scheduling multithreaded computations by work stealing. In: FOCS, pp. 356–368. IEEE Computer Society (1994)
8. Bouali, A., de Simone, R.: Symbolic bisimulation minimisation. In: Computer Aided Verification, 4th International Workshop, LNCS, vol. 663, pp. 96–108. Springer (1992)
9. Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient implementation of a BDD package. In: DAC, pp. 40–45 (1990)
10. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **C–35**(8), 677–691 (1986)
11. Burch, J., Clarke, E., Long, D., McMillan, K., Dill, D.: Symbolic model checking for sequential circuit verification. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **13**(4), 401–424 (1994)
12. Clarke, E.M., McMillan, K.L., Zhao, X., Fujita, M., Yang, J.: Spectral transforms for large Boolean functions with applications to technology mapping. In: DAC, pp. 54–60 (1993)
13. Cranen, S., Groote, J.F., Keiren, J.J.A., Stappers, F.P.M., de Vink, E.P., Wesselink, W., Willemse, T.A.C.: An overview of the mCRL2 toolset and its recent advances. In: TACAS, LNCS, vol. 7795, pp. 199–213. Springer (2013)
14. De Nicola, R., Vaandrager, F.W.: Three logics for branching bisimulation. J. ACM **42**(2), 458–487 (1995)
15. Derisavi, S.: A symbolic algorithm for optimal Markov chain lumping. TACAS **2007**, 139–154 (2007)
16. Derisavi, S.: Signature-based symbolic algorithm for optimal Markov chain lumping. In: QEST 2007, pp. 141–150. IEEE Computer Society (2007)
17. van Dijk, T.: Sylvan: multi-core decision diagrams. Ph.D. thesis, University of Twente (2016)
18. van Dijk, T., Laarman, A., van de Pol, J.: Multi-core BDD operations for symbolic reachability. ENTCS **296**, 127–143 (2013)
19. van Dijk, T., van de Pol, J.: Lace: non-blocking split deque for work-stealing. In: MuCoCoS, LNCS, vol. 8806, pp. 206–217. Springer (2014)
20. van Dijk, T., van de Pol, J.: Sylvan: multi-core decision diagrams. In: TACAS, LNCS, vol. 9035, pp. 677–691. Springer (2015)
21. van Dijk, T., van de Pol, J.: Multi-core symbolic bisimulation minimisation. In: TACAS, LNCS, vol. 9636, pp. 332–348. Springer (2016)
22. Dolev, D., Klawe, M.M., Rodeh, M.: An $o(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. J. Algorithms **3**(3), 245–260 (1982)
23. Franklin, W.R.: On an improved algorithm for decentralized extrema finding in circular configurations of processors. Commun. ACM **25**(5), 336–337 (1982)
24. Groote, J.F., van de Pol, J.: A bounded retransmission protocol for large data packets. In: Wirsing, M., Nivat, M. (eds.) AMAST'96, LNCS 1101, pp. 536–550. Springer, Berlin (1996)
25. Hermanns, H.: Interactive Markov Chains: The Quest for Quantified Quality, Lecture Notes in Computer Science, vol. 2428. Springer, Berlin (2002)
26. Hermanns, H., Katoen, J.: The how and why of interactive Markov chains. In: FMCO'09, LNCS 6286, pp. 311–337. Springer (2009)
27. Hesselink, W.H.: Invariants for the construction of a handshake register. Inf. Process. Lett. **68**(4), 173–177 (1998)
28. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: TACAS 2015, LNCS, vol. 9035, pp. 692–707. Springer (2015)
29. Kulakowski, K.: Concurrent bisimulation algorithm. CoRR. arXiv:1311.7635 (2013)
30. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: CAV, LNCS, vol. 6806, pp. 585–591. Springer (2011)
31. Meijer, J., Kant, G., Blom, S., van de Pol, J.: Read, write and copy dependencies for symbolic model checking. In: Yahav, E. (ed.) HVC, LNCS, vol. 8855, pp. 204–219. Springer, Berlin (2014)
32. Mumme, M., Ciardo, G.: An efficient fully symbolic bisimulation algorithm for non-deterministic systems. Int. J. Found. Comput. Sci. **24**(2), 263–282 (2013)
33. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. **16**(6), 973–989 (1987)
34. van de Pol, J., Timmer, M.: State space reduction of linear processes using control flow reconstruction. In: Liu, Z., Ravn, A.P. (eds.) ATVA'09, LNCS, vol. 5799, pp. 54–68. Springer, Berlin (2009)
35. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Commun. ACM **33**(6), 668–676 (1990)
36. Remenska, D., Willemse, T.A.C., Verstoep, K., Fokkink, W., Templon, J., Bal, H.E.: Using model checking to analyze the system behavior of the LHC production grid. In: CCGrid'12, pp. 335–343. IEEE Computer Society (2012)
37. Wijs, A.: GPU accelerated strong and branching bisimilarity checking. TACAS **2015**, 368–383 (2015)
38. Wimmer, R., Becker, B.: Correctness issues of symbolic bisimulation computation for Markov chains. In: MMB&DFT, LNCS, vol. 5987, pp. 287–301. Springer (2010)
39. Wimmer, R., Derisavi, S., Hermanns, H.: Symbolic partition refinement with automatic balancing of time and space. Perform. Eval. **67**(9), 816–836 (2010)

40. Wimmer, R., Herbstritt, M., Becker, B.: Optimization techniques for BDD-based bisimulation computation. In: 17th GLSVLSI, pp. 405–410. ACM (2007)
41. Wimmer, R., Herbstritt, M., Hermanns, H., Strampp, K., Becker, B.: Sigref—a symbolic bisimulation tool box. In: ATVA, LNCS, vol. 4218, pp. 477–492. Springer (2006)
42. Wimmer, R., Hermanns, H., Herbstritt, M., Becker, B.: Towards symbolic stochastic aggregation. Technical Report, SFB/TR 14 AVACS (2007)