Symbolic Partition Refinement with Automatic Balancing of Time and Space[☆]

Ralf Wimmer

Institute of Computer Science, Albert-Ludwigs-University Freiburg, Germany

Salem Derisavi

IBM Toronto Software Lab, Canada[☆☆]

Holger Hermanns

Department of Computer Science, Saarland University, Germany

Abstract

State space lumping is one of the classical means to fight the state space explosion problem in state-based performance evaluation and verification. Particularly when numerical algorithms are applied to analyze a Markov model, one often observes that those algorithms do not scale beyond systems of moderate size. To alleviate this problem, symbolic lumping algorithms have been devised to effectively reduce very large – but symbolically represented – Markov models to moderate size explicit representations. This lumping step partitions the Markov model in such a way that any numerical analysis carried out on the lumped model is guaranteed to produce exact results for the original system. But even this lumping preprocessing may fail due to time or memory limitations. This paper discusses the two main approaches to symbolic lumping, and combines them to improve on their respective limitations. The algorithm automatically converts between known symbolic partition representations in order to provide a trade-off between memory consumption and runtime. We show how to apply this algorithm for the lumping of Markov chains, but the same techniques can be adapted in a straightforward way to other models like Markov reward models, labeled transition systems, or interactive Markov chains.

Key words: state space lumping, symbolic methods, Markov chains

1. Introduction

Markov chains are among the most fundamental mathematical structures used for performance and dependability modeling of communication and computer systems. Since the size of a Markov chain usually grows exponentially with the size of the corresponding high-level model, one often encounters the state space explosion problem, which frequently makes the analysis of the Markov chain intractable.

In the area of formal verification, bisimulation equivalence [1] plays a prominent role as an equivalence relation on state transition graphs. It equates two states if and only if their future behavior is indistinguishable. In the context of Markov models, the same idea is known as *lumpability* [2]. Originally, lumpability was

[☆]This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See `http://www.avacs.org` for more information.

[☆☆]Part of this work was done while the co-author was at Carleton University in Ottawa, Canada.

Email addresses: `wimmer@informatik.uni-freiburg.de` (Ralf Wimmer), `derisavi@ca.ibm.com` (Salem Derisavi), `hermanns@cs.uni-sb.de` (Holger Hermanns)

defined with respect to a given partition of the state space. If the lumpability condition is satisfied for this partition, an often much smaller model can be obtained by considering the quotient induced by the partition. That quotient, the *lumped* Markov chain, captures all relevant behavior.

Many approaches to alleviate or circumvent the state space explosion problem for Markov chains are implicitly or explicitly based on the notion of lumpability, since this allows computation of measures of the original Markov chain using the analysis of the lumped Markov chain. A core practical challenge in this context is to devise a proper partition of the state space ensuring that the lumping conditions hold, while avoiding to actually generate the (possibly excessively large) state space representation of the system to be lumped.

One strand of work exploits information available in the high-level model, such as symmetries or hierarchies, in order to directly generate the lumped model. Examples of such model-level approaches include [3, 4, 5, 6, 7, 8]. Another strand of work is a result of looking at lumpability from the bisimulation perspective [9, 10, 11]. It allows us to formulate lumpability as a fixpoint of a higher order function on the original state space, and gives rise to an effective algorithm to compute the *smallest possible* partition of a Markov chain that satisfies the lumping conditions. This means that instead of using high-level model information, which is cheaper but may lead to sub-optimal lumping, the optimally lumped Markov chain can be computed by an efficient fixpoint algorithm [12, 13]. Since this can in some cases be done faster than doing the actual analysis on the original model, it is possible to preprocess an arbitrary Markov chain to derive the best lumped Markov chain, and then proceed with the analysis of the latter. Due to the reduction in state space size, this two-step approach often speeds up the overall analysis time [14].

While this appears as a great step forward, the optimal lumping approach does not solve the state space explosion problem *per se*, since the fixpoint algorithm runs on the original, possibly excessively large state space. This is where symbolic representations come into play: Instead of explicitly representing the original state space, it is represented in a symbolic manner, using structures such as BDDs or MTBDDs [15, 16, 17, 18, 19, 20, 21, 22, 23]. These structures are equipped with well-understood heuristics such that in most cases they only grow moderately in the size of the high-level compositional model [24, 25, 26].

Notably, numerical analysis techniques do not scale if directly applied on symbolic representations. As observed in [26, 27], the MTBDD representation of the solution vector tends to grow extremely large, despite a compact MTBDD representation of the model. This is caused by lacking regularity and diversity of values in the solution vector as the computation progresses. Resorting to EVBDDs instead of MTBDDs can lead to a similar blow up during model checking, not in terminal nodes, but in edge value labelings, induced by the irregularity of the values and operations occurring. However, symbolic representations play a key role if used to represent the original state space in order to then generate the best possible lumping.

These considerations motivate the recent work on symbolic algorithms for optimal lumping. Rooted in the work of Blom and Orzan [28, 29, 30], who developed a distributed algorithm for bisimulation minimization, different symbolic lumping algorithms have been developed [31, 32, 33]. In combination with compositional construction techniques, they have been applied to models of sizes otherwise far out of reach of contemporary numerical analysis engines [23, 34].

While the above symbolic algorithms are similar in spirit, they have conceptual and practically relevant differences. The work of [31] utilizes a "fast" partition representation that makes it possible to have a very efficient algorithm in terms of computation time, but even for fairly small numbers of equivalence classes it consumes a considerable amount of memory. On the other hand, the "compact" partition representation of [32] stays very small in terms of space requirements, but its drawback is that performing operations on the representation can become quite expensive timewise. The difference is caused by drastically different representation techniques for encoding the state space partitions as BDDs, which are accessed and refined by the algorithm.

To further push the limits of this technology, this paper provides an in-depth study of the earlier approaches to symbolic lumping and then devises a combination of them. We develop an algorithm which is memory efficient by using the compact partition representation of [32] and runtime efficient by using the fast partition representation and refinement algorithm of [31]. Our "hybrid" approach offers a "spectrum" of representations whose extremes are the fast representation on one side and the compact representation on the other. It provides us a parameter by which we can control where in the spectrum a specific instance of the representation stands.

In principle, it is possible to implement our algorithms using both MTBDDs and EVBDDs. However, we do not expect great reductions in memory consumption by using EVBDDs, because the central data structure for representing partitions of state spaces is a vector of Boolean functions. For Boolean functions the size of BDDs and EVBDDs is identical up to one node [18]. Since our algorithmic ideas are more easy to explain in the context of MTBDDs and since efficient and well-tested implementations of MTBDDs are available, we use MTBDDs for the presentation and the experiments.

The contributions of the paper are 1) an algorithm that converts between fast and compact partition representations in a logarithmic number of BDD operations, 2) a simple but effective algorithm that automatically changes the parameter mentioned above to balance the time and space requirements of the algorithm such that the refinement works at maximal speed without exceeding the available memory, and 3) an implementation of the conversion and parameter selection algorithms into the principal refinement algorithm. We experimentally evaluate the benefits of our algorithm, and compare its performance with the algorithms of [33] (that uses the fast representation of [31]) and [32].

The entire work is presented here in the context of continuous-time Markov chain lumping. However, there is a much broader spectrum of possible applications, since the techniques are straightforwardly adaptable to labeled transition systems, discrete-time or interactive Markov chains, Markov reward models, Markov decision processes etc. The core contribution of this paper is thus a general, fast, and memory efficient algorithm for symbolic lumping and symbolic bisimulation minimization.

Organization of the paper: Section 2 reviews the basic concepts and the context in which the present paper is placed. In Section 3, we discuss the principal algorithmic considerations behind a non-symbolic lumping algorithm. After discussing the earlier approaches to symbolic lumping in Section 4, Section 5 introduces the new hybrid algorithm. Experimental results are presented in Section 6 demonstrating the effectiveness of our approach. Section 7 concludes the paper.

2. Background

In this section, we first review the concepts of continuous-time Markov chains and lumpability. We then give a brief account of the general principle of signature-based lumping algorithms. We finally review symbolic data structures and their use to represent various entities appearing in our context, such as sets and matrices.

2.1. Markov chains and state space lumping

A *continuous-time Markov chain* (MC) M is a pair $M = (S, \mathbf{R})$ where S is a finite non-empty set of states and $\mathbf{R} : S \times S \to \mathbb{R}^{\geq 0}$ is the transition rate matrix such that $\mathbf{R}(s,s) = 0$ for all $s \in S$. The *generator matrix* $\mathbf{Q} : S \times S \to \mathbb{R}$ is defined as $\mathbf{Q}(s,s) = -\sum_{s' \in S} \mathbf{R}(s,s')$ and $\mathbf{Q}(s,t) = \mathbf{R}(s,t)$ for all $s,t \in S$ with $s \neq t$.

A *partition* P of a set S is a set of pairwise disjoint, non-empty subsets of S, such that their union equals S. Elements of P are called *blocks* of P. In the sequel, a *subpartition* is a subset of a partition. For elements s and t in the same block of P, we write $s \equiv_P t$. The block of P which contains $s \in S$ is denoted by $[s]_P$. Let P and P' be partitions of S. P is called a *refinement* of P' (or conversely P' coarser than P), denoted by $P \sqsubseteq P'$, if $\forall B \in P \ \exists B' \in P' : B \subseteq B'$.

For a matrix $\mathbf{A} \in \mathbb{R}^{|S| \times |S|}$ and subsets $B, B' \subseteq S$, we define $\mathbf{A}(B,B') = \sum\limits_{s \in B} \sum\limits_{s' \in B'} \mathbf{A}(s,s')$. For $s \in S$ and $B \subseteq S$, we use $\mathbf{A}(s,B)$ and $\mathbf{A}(B,s)$ instead of $\mathbf{A}(\{s\},B)$ and $\mathbf{A}(B,\{s\})$ respectively.

Definition 1. Let $M = (S, \mathbf{R})$ be an MC with generator matrix $\mathbf{Q}$. With respect to a partition P of S, M is

- *ordinarily lumpable* if $\forall C, C' \in P \ \forall s, s' \in C : \mathbf{Q}(s, C') = \mathbf{Q}(s', C')$, and
- *exactly lumpable* if $\forall C, C' \in P \ \forall s, s' \in C : \mathbf{Q}(C', s) = \mathbf{Q}(C', s')$,
- *strictly lumpable* if it is ordinarily and exactly lumpable with respect to P.

We recall briefly [12, 35, 36] in what sense each variation of lumpability preserves the behavior of the Markov chain.

Ordinary lumpability ensures that the probability distribution at any time point of being in any particular partition equals the sum of the state probabilities in that partition at the given time. Therefore we can shrink the Markov chain by replacing each block of the partition by a single state, provided we are only interested in the probabilities of entire blocks.

Exact lumpability fulfills a different interesting property: Starting with an initial distribution that is equally distributed inside each block, the distribution at any time stays equally distributed inside each block. More formally: Let $p(s, s', t)$ denote the probability that control is in state s' at time t, given the system was started in state s at time 0; let $p_0 : S \to [0, 1]$ denote the probability distribution which assigns each state the probability of being chosen as the initial state. Furthermore, let P^0 be an initial partition such that for all $s, s' \in S : p_0(s) \neq p_0(t) \Rightarrow s \not\equiv_{P^{(0)}} s'$ and P be an exactly lumpable partition with $P \sqsubseteq P^{(0)}$. Then, $p(s, s', t) = p(s, s'', t)$ holds for all $t \geq 0$ and all $s', s'' \in S$ with $s' \equiv_P s''$. Therefore, we can shrink the Markov chain by replacing each block of the partition by a single state, and still retrieve the individual state probabilities, provided that the equal distribution condition on the initial distribution is met. Strict lumpability induces both the above properties. Not surprisingly, those properties also hold for the steady-state limiting distributions.

In the following, we restrict ourselves to ordinary lumping, and postpone the discussion on how to handle exact and strict lumpability to Section 3.3. A partition P of an MC satisfying the conditions of ordinary lumpability in Definition 1 will be called a *lumpable partition.*

Definition 2. Let $M = (S, \mathbf{R})$ be an MC that is ordinarily lumpable with respect to a partition P of S. Then $\tilde{M} = (\tilde{S}, \tilde{\mathbf{R}})$ is the *lumped (or, quotient) MC* such that

$$\tilde{S} = \{\text{arbitrary element of } C \mid C \in P\}$$

$$\tilde{\mathbf{R}}(\tilde{s}, \tilde{s}') = \begin{cases} \mathbf{R}(\tilde{s}, [\tilde{s}']_P) & \text{if } \tilde{s} \neq \tilde{s}' \\ 0 & \text{if } \tilde{s} = \tilde{s}'. \end{cases}$$

Note that although $\tilde{S}$ apparently depends on the arbitrarily selected element of each class of P, the conditions in Definition 1 ensure that all possible lumped MCs are isomorphic (identical up to state identities).

2.2. (Multi-terminal) Binary Decision Diagrams

Since the amount of memory needed by an explicit representation of the state space (e. g. by a sparse matrix representation) is linear in the size of the represented system, the available memory severely limits the size of the systems tractable using explicit techniques.

The strength of symbolic representations like matrix diagrams (MDs) [20], and different flavors of decision diagrams such as (Multi-terminal) Binary Decision Diagrams, (MT)BDDs [15, 16], edge-valued binary decision diagrams (EVBDDs) [17], probabilistic decision graphs (PDGs) [37], and zero-suppressed decision diagrams (ZBDDs) [19, 38] is that the size of the representation does not grow as excessive as the size of the represented systems. In practice, by using various heuristics the size of the representation often becomes much smaller and grows about linearly in the size the high-level model, instead of growing linearly with the state space size [24]. Dedicated algorithms which exploit the structure of this compact representation can often handle very large systems, because often their runtime is approximately linear in the size of that representation [15, 16, 39]. We work with MTBDDs in this paper.

Definition 3. Let $x = (x_1, \ldots, x_h)$ be a vector of Boolean variables. A *multi-terminal binary decision diagram* (MTBDD) $\mathcal{G}(x) = (V, r, E)$ is an acyclic, directed graph with root $r \in V$ such that the following conditions hold:

- Each node $v \in V$ is either an inner node or a leaf.
- Leaves $v \in V$ do not have outgoing edges and are labeled with a real value $\mathsf{label}(v) \in \mathbb{R}$.

- Inner nodes $v \in V$ have exactly two successor nodes, denoted $\mathsf{low}(v)$ and $\mathsf{high}(v)$. Inner nodes are labeled by a variable $\mathsf{label}(v) \in \{x_1, \dots, x_h\}$.

If all leaves are labeled only with 0 and 1, $\mathcal{G}$ is called a BDD. Each node $v \in V$ of an MTBDD represents a function $f_v : \{0,1\}^h \to \mathbb{R}$, which is defined as follows:

Definition 4. Let $\mathcal{G}(x) = (V, r, E)$ be an MTBDD. We assign each node $v \in V$ a function $f_v : \{0,1\}^h \to \mathbb{R}$ as follows:

- If v is a leaf node, then $f_v(x) = \mathsf{label}(v)$.
- If v is an inner node with $\mathsf{label}(v) = x_i$, then

$$f_v(x) = x_i \cdot f_{\mathsf{high}(v)}(x) + (1 - x_i) \cdot f_{\mathsf{low}(v)}(x).$$

The function represented by $\mathcal{G}(x)$ is the function f_r assigned to the root node.

In order to obtain a data structure which can algorithmically be handled efficiently, we need further restrictions:

Let $\mathcal{G}(x) = (V, r, E)$ be an MTBDD. $\mathcal{G}(x)$ is *free* if on each path from r to a leaf node each variable occurs at most once as node label. It is called *ordered* if $\mathcal{G}(x)$ is free and on each path from r to a leaf node, the variables occur in the same order. Finally, $\mathcal{G}(x)$ is *reduced* if $\forall u, v \in V : f_u \neq f_v$.

From now on, we will assume, that all (MT)BDDs are ordered and reduced. We will denote them by calligraphic letters. We refer to the number of nodes of an MTBDD $\mathcal{G}(x)$ by $|\mathcal{G}(x)|$ and call it the size of $\mathcal{G}(x)$. In the following, we explain how to use (MT)BDDs to represent sets and matrices. When clear from the context, we will furthermore identify an (MT)BDD with the represented function and omit the variables which occur as labels of the BDD nodes.

Set representation. Let $N \subseteq \{0,1\}^h$ be a set of Boolean vectors. N can be represented symbolically by a BDD $\mathcal{N}(x)$ such that $\mathcal{N}(x) = 1$ if $x \in N$ and 0 otherwise.

Matrix representation. We use MTBDDs to efficiently represent transition matrices of MCs. A matrix $\mathbf{R} : \{0,1\}^h \times \{0,1\}^h \to \mathbb{R}$ can be represented using an MTBDD $\mathcal{R}$ with $2h$ binary variables. The first h variables encode the row index and the other h variables the column index. We use an interleaved variable ordering in which each row variable is immediately followed by its corresponding column variable or vice versa. An interleaved variable ordering often leads to small MTBDDs for MCs which are generated from high-level models [40].

3. Explicit-State Lumping Algorithms

In the following, we present two different algorithms which compute the coarsest lumping quotient that refines an initial partition $P^{(0)}$, possibly induced by atomic labels, rewards, etc. attached to states. If no initial partition is explicitly given, we set $P^{(0)} = \{S\}$.

3.1. Traditional Lumping Algorithm

The basis for most explicit-state lumping algorithms is the algorithm described in [13]. It is optimal for ordinary lumping and has a running time of $O(m \cdot \log n)$, where n is the number of states and m the number of transitions of the Markov chain with a non-zero rate.

Algorithm 3.1 shows the pseudo-code of this explicit-state lumping algorithm. LumpMC takes the original MC M and returns the quotient MC $\tilde{M}$. It works in two stages. First, LumpablePart computes the coarsest partition P with respect to which M is lumpable by iterative refinements of the initial partition $P^{(0)}$, until a fixpoint is reached. In the second stage (line 2), CompQuotient (whose pseudo-code is not shown here) computes the quotient $\tilde{M}$ according to Definition 2.

```
LumpMC($M, P^{(0)}$)
1 $P :=$ LumpablePart($S, \mathbf{R}, P^{(0)}$)
2 $(\tilde{S}, \tilde{\mathbf{R}}) :=$ CompQuotient($S, \mathbf{R}, P$)
3 return $\tilde{M} := (\tilde{S}, \tilde{\mathbf{R}})$
```

```
LumpablePart($S, \mathbf{R}, P^{(0)}$)
1 $P := P^{(0)}$
2 $L := P^{(0)}$
3 while $L \neq \emptyset$
4     $B :=$ Pop($L$)
5     Split($P, B, L$)
6 return $P$
```

```
Split($P, B, L$)
1 $P_{\text{old}} := P$
2 foreach $C \in P_{\text{old}}$
3     $\{C_1, \dots, C'_\alpha\} := \big\{\{s \in C \mid \mathbf{R}(s, B) = \mathbf{R}(s', B)\} \bigm| s' \in C\big\}$
4     $P := \big(P \setminus \{C\}\big) \dot\cup \{C_1, \dots, C'_\alpha\}$
5     $L := L \dot\cup \big(\{C_1, \dots, C'_\alpha\} \setminus \text{largest } C'_i\big)$
```

Algorithm 3.1: Runtime-optimal explicit-state lumping algorithm

LumpablePart maintains a list L of potential *splitters*. Each refinement iteration of LumpablePart (lines 3–5) refines P with respect to one potential splitter B. Split splits each class C of P into classes $C'_1, \dots C'_\alpha$ (line 3) by grouping the states of C based on their cumulative outgoing rates to B. More formally:

$$\textit{refinement}(C, B) = \big\{\{s \in C \mid \mathbf{R}(s, B) = \mathbf{R}(s', B)\} \bigm| s' \in C\big\}. \tag{1}$$

In line 5 of Split, the list of potential splitters is updated. We need to add all newly generated blocks but one to the list. We can neglect any single block C'_i, because its power of splitting other blocks is maintained by the remaining sub-blocks. Excluding the largest sub-block C'_i from the set $\{C'_1, \dots, C'_\alpha\}$ of potential splitters is similar to the "process the smaller half" strategy given by Hopcroft [41]. It has been proven that the algorithm runs in time $O(m \cdot \log n)$ when using this strategy [13].

The algorithm finishes when P is refined with respect to all potential splitters. The result is the coarsest ordinarily lumpable partition that is a refinement of $P^{(0)}$. See [13, 42] for more details.

3.2. Markov Chain Lumping using Signature-based Refinement

Most lumping algorithms in the literature use – like the algorithm presented in the previous section – iterative partition refinement such that, in each iteration, the current partition is refined with respect to a block retrieved from a list of potential splitters. Blom and Orzan were the first to devise an iterative algorithm that, in each iteration, refines the current partition with respect to *all* blocks simultaneously [28]. Their algorithm works by computing, in each iteration, the *signature* of all states with respect to the current partition (as opposed to the current splitter in conventional algorithms). Defined formally in Eq. (2), the signature of a state with respect to a partition is the total transition rate from the state to each block of the partition. In each refinement step, states are kept in the same block iff they have the same signature. The algorithm stops once the partition reaches a fixpoint, i. e., a partition that will not be split any further.

This signature-based principle is independent of the type of representation (i. e., explicit or symbolic) used. While the idea was originally used to design an explicit and distributed algorithm for branching bisimulation minimization of non-probabilistic transition systems, Wimmer et al. [31] and Derisavi [33] used it to develop *symbolic* lumping algorithms for non-probabilistic and probabilistic systems, respectively.

For a given MC $M = (S, \mathbf{R})$ with generator matrix $\mathbf{Q}$ and initial partition $P^{(0)}$ of S, the signature-based algorithm to compute the coarsest lumpable partition refining $P^{(0)}$ is given in Algorithm 3.2. The actual refinement is done by the sigref-operator, which is defined as follows:

$$\begin{aligned}\mathsf{sigref}(P) &= \big\{\{s \in S \mid \mathsf{sig}(P, s) = \mathsf{sig}(P, t) \wedge s \equiv_{P^{(0)}} t\} \bigm| t \in S\big\}\\ \mathsf{sig}(P, s) &= \big\{(r, B) \in \mathbb{R} \times P \bigm| r = \mathbf{Q}(s, B)\big\}\end{aligned} \tag{2}$$

Starting with the given initial partition $P^{(0)}$, the algorithm iteratively applies the sigref-operator until a fixpoint is reached. Theorem 1 guarantees that the fixpoint is the coarsest partition of S with respect to which the MC is lumpable. The quotient system can be extracted in the same way as in Algorithm 3.1.

```
LumpablePartSigBased($M, P^{(0)}$)
1  $i := 0$
2  do
3      $P^{(i+1)} := \mathsf{sigref}(P^{(i)})$ according to Eq. (2)
4      $i := i + 1$
5  until $P^{(i)} = P^{(i-1)}$
6  return $P^{(i)}$
```

Algorithm 3.2: Explicit-state signature-based lumping algorithm

Theorem 1. *Let $M = (S, \mathbf{R})$ be an MC and $P^{(0)}, P^{(1)}, \ldots$ be a sequence of partitions of S with $P^{(i+1)} = \mathsf{sigref}(P^{(i)})$ for $i \geq 0$. There exists $f \leq |S| - |P^{(0)}|$ such that $P^{(f+1)} = P^{(f)}$ and $P^{(f)}$ is the coarsest refinement of $P^{(0)}$ with respect to which M is lumpable.*

Proof. First we prove that $P^{(n+1)} \sqsubseteq P^{(n)}$ for all $n \geq 0$. This implies the existence of a fixpoint after at most $|S| - |P^{(0)}|$ splitting steps, because the sequence of partitions is monotonic and bounded below by $\big\{\{s\} \mid s \in S\big\}$. Additionally, in each step the number of blocks increases by at least one until a fixpoint is reached.

We show our claim by induction on n. The base case ($n = 0$) holds, because by definition of sigref, each partition which is created by sigref is a refinement of $P^{(0)}$. Now assume that $P^{(i)} \sqsubseteq P^{(i-1)}$ for a given $i > 0$. We will now show that this implies $P^{(i+1)} \sqsubseteq P^{(i)}$, or equivalently $\forall s, t \in S : s \equiv_{P^{(i+1)}} t \Rightarrow s \equiv_{P^{(i)}} t$. Given $s, t \in S$, $s \equiv_{P^{(i+1)}} t$ implies that $\mathsf{sig}(P^{(i)}, s) = \mathsf{sig}(P^{(i)}, t)$, and therefore, $\forall B \in P^{(i)} : \mathbf{Q}(s, B^{(i)}) = \mathbf{Q}(t, B^{(i)})$. Since $P^{(i)} \sqsubseteq P^{(i-1)}$, any block $B^{(i-1)} \in P^{(i-1)}$ is the union of a number of pairwise disjoint blocks of $P^{(i)}$, i. e. $B^{(i-1)} = \bigcup_{j=1}^{b} B_j^{(j)}$ with $B_j^{(i)} \in P^{(i)}$. Therefore, we have $\mathbf{Q}(s, B^{(i-1)}) = \sum_{j=1}^{b} \mathbf{Q}(s, B_j^{(i)}) = \sum_{j=1}^{b} \mathbf{Q}(t, B_j^{(i)}) = \mathbf{Q}(s, B^{(i-1)})$ which implies $s \equiv_{P^{(i-1)}} t$.

The next step is to show that each fixpoint of sigref induces a lumpable partition. Let P be a fixpoint of sigref, i. e. $P = \mathsf{sigref}(P)$. We have for all $s, t \in S$ with $s \equiv_P t$ that $\mathsf{sig}(s, P) = \mathsf{sig}(t, P)$ holds. This implies $\forall B \in P : \mathbf{Q}(s, B) = \mathbf{Q}(t, B)$. Hence, P is a lumpable partition, a partition satisfying the conditions on ordinary lumpability in Definition 1.

The last point we have to prove is that the fixpoint $P^{(f)}$ reached by the algorithm is the coarsest lumpable partition which refines $P^{(0)}$. Let P^* be a lumpable partition with $P^* \sqsubseteq P^{(0)}$. We show that $\forall i \geq 0 : P^* \sqsubseteq P^{(i)}$, from which our claim follows. We prove this by induction on i. Obviously, the base case $i = 0$ holds. Now assume $P^* \sqsubseteq P^{(i)}$ for some $i \geq 0$. Therefore, any $B^{(i)} \in P^{(i)}$ is the union of a number of pairwise disjoint blocks of P^*, i. e. $B^{(i)} = \bigcup_{j=1}^{b} B_j^*$ with $B_j^* \in P^*$. We prove that $P^* \sqsubseteq P^{(i+1)}$, or equivalently $\forall s, t \in S : s \equiv_{P^*} t \Rightarrow s \equiv_{P^{(j+1)}} t$. Let $s, t \in S$ be states such that $s \equiv_{P^*} t$. Using the induction hypothesis, we have that $s \equiv_{P^{(i)}} t$. Since P^* is a stochastic bisimulation, we have that $\mathbf{Q}(s, B_j^*) = \mathbf{Q}(t, B_j^*)$. Therefore $\mathbf{Q}(s, B^{(i)}) = \sum_{j=1}^{b} \mathbf{Q}(s, B_j^*) = \sum_{j=1}^{b} \mathbf{Q}(t, B_j^*) = \mathbf{Q}(t, B^{(i)})$. For this reason, the signatures of s and t with respect to $P^{(i)}$ are identical. So $s \equiv_{P^{(i+1)}} t$. $\square$

3.3. Discussion

The runtime of the latter algorithm is in $O(m \cdot n)$ where n is the number of states and m the number of transitions with a non-zero rate. In contrast, the former algorithm can be implemented such that it runs in $O(m \cdot \log n)$ time. However, these theoretical worst case bounds do not say much about their practical runtime [42], in particular in the symbolic context we are studying in the remainder of this paper.

Before we go into details of the symbolic implementation of these algorithms, we briefly comment on the open question of how to compute optimal partitions with respect to exact and strict lumpability with either of these algorithms: Exact lumping can by reduced to ordinary lumping by reversing the direction of all edges in the Markov chain prior to the algorithm, and reversing them again after quotient construction. To compute the optimal partition with respect to strict lumpability, the entire fixpoint computation for ordinary and exact lumpability are applied alternately until an overall fixpoint is reached, i. e. the partition does not change anymore.

4. Symbolic Lumping Algorithms

In order to obtain an efficient symbolic algorithm which computes the coarsest lumpable partition of a Markov chain, we first have to find an appropriate symbolic partition representation. Requirements are not only compactness but also an efficient support of the necessary operations. The most common techniques will be presented in this section. Then we will show how the two explicit-state lumping algorithms from Section 3 can be turned into symbolic algorithms. Their advantages and disadvantages are discussed afterward.

4.1. Partition Representations

The representation of state space partitions appearing in the refinement algorithm is a crucial aspect of the setting considered in this paper. We are aware of four distinct techniques for the symbolic representation of a partition $P = \{B_0, \ldots, B_{n-1}\}$. For the third and fourth technique, we presuppose an arbitrary, but fixed order on the blocks of each represented partition.

1. To use a BDD $\mathcal{P}_{\mathrm{ER}}$ to represent the corresponding equivalence relation $\equiv_P$ such that $\mathcal{P}_{\mathrm{ER}}(s,t) = 1$ iff $s \equiv_P t$, i. e. iff $\exists B_i \in P : s \in B_i \wedge t \in B_i$. This representation is used, e. g., in [43], which pioneered symbolic bisimulation minimization. (ER)
2. To use one BDD per block. A partition representation is then a set $\{\mathcal{P}^0_{\mathrm{BR}}, \ldots, \mathcal{P}^{n-1}_{\mathrm{BR}}\}$ of BDDs such that $\mathcal{P}^i_{\mathrm{BR}}(s) = 1$ iff $s \in B_i$. (BR)
3. To use an extra vector k of at least $\lceil \log_2 n \rceil$ new BDD variables to denote the block index. For the block index a binary encoding is used. The partition is then represented by a BDD $\mathcal{P}_{\mathrm{FR}}$ such that $\mathcal{P}_{\mathrm{FR}}(s,k) = 1$ iff $s \in B_k$. To access the states of a block, a single cofactor computation with respect to the block variables is necessary, i. e.

   $$\mathcal{B}_i(s) = \mathcal{P}_{\mathrm{FR}}(s,k)_{|k=i}. \tag{3}$$

   This representation was introduced in [31] to compute various kinds of bisimulation on labeled transition systems, and adopted in [33] for Markov chain lumping. (FR)
4. To use a vector of $d = \lceil \log_2 n \rceil$ BDDs $(\mathcal{P}^0_{\mathrm{CR}}, \ldots, \mathcal{P}^{d-1}_{\mathrm{CR}})$ such that $\mathcal{P}^j_{\mathrm{CR}}(s) = 1$ iff $s \in B_i$ and the j^{th} bit of i is one. In other words, $\mathcal{P}^j_{\mathrm{CR}}$ is the union of all blocks whose indices have 1 in their j^{th} bit, i. e.

   $$\mathcal{P}^j_{\mathrm{CR}}(s) = \bigvee_{\substack{0 \le i < n \\ \text{the } j^{\text{th}} \text{ bit of } i \text{ is } 1}} \mathcal{B}_i(s) \tag{4}$$

   Restoring the block B_i from this representation uses $O(\log n)$ BDD operations as follows:

   $$\mathcal{B}_i(s) = \bigwedge_{\substack{0 \le j < d \\ \text{the } j^{\text{th}} \text{ bit of } i \text{ is } 1}} \mathcal{P}^j_{\mathrm{CR}}(s) \quad \wedge \bigwedge_{\substack{0 \le j < d \\ \text{the } j^{\text{th}} \text{ bit of } i \text{ is } 0}} (\mathcal{S}(s) \wedge \neg \mathcal{P}^j_{\mathrm{CR}}(s)). \tag{5}$$

   This representation was introduced in [32], for symbolic computation of lumpability in Markov chains, which we will briefly describe in the next section. (CR)

For a partition P, represented symbolically by (one or more) MTBDDs $\mathcal{P}$ using one of the four possibilities introduced above, we denote the number of blocks of the partition by $\#\mathcal{P}$, i. e. $\#\mathcal{P} = |P|$.

We provide a small example to illustrate the latter two partition representation techniques CR and FR, which will play a central role in this paper.

Example 1. *Let $P = \{B_0, B_1, B_2, B_3\}$ be a partition of $S = \{s_0, s_1, \ldots, s_8\}$ consisting of four blocks*

$$B_0 = \{s_0, s_3, s_4\} \qquad B_1 = \{s_1, s_2, s_7\}$$
$$B_2 = \{s_5, s_8\} \qquad B_3 = \{s_6\}.$$

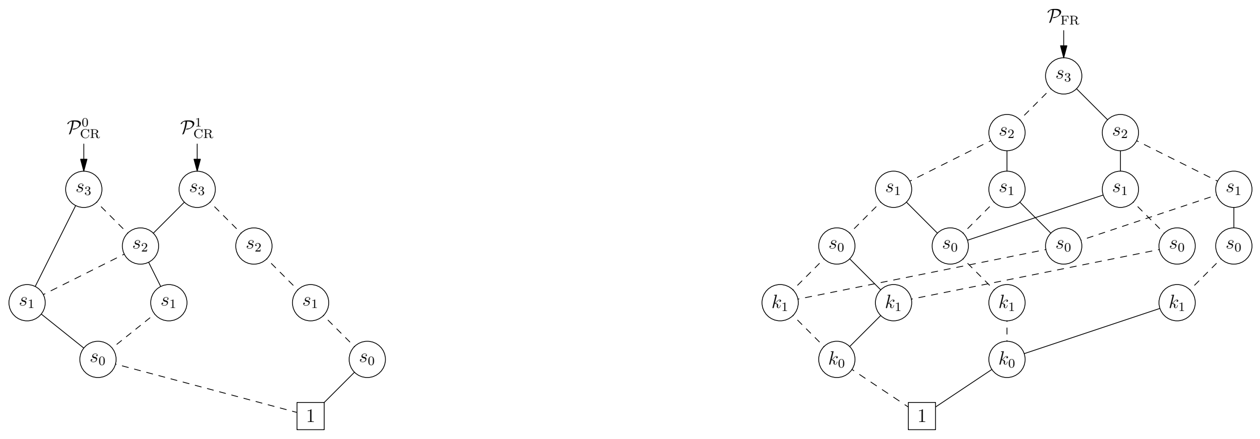

$\mathcal{P}^0_{\mathrm{CR}}$
$\mathcal{P}^1_{\mathrm{CR}}$
$\mathcal{P}_{\mathrm{FR}}$
s_3
s_2
s_1
s_0
k_1
k_0
1

Figure 1: CR (left) vs. FR (right) representation

- *The* FR*-representation uses one BDD* $\mathcal{P}_{\mathrm{FR}}(s,k)$ *representing the set*

$$\{(s_0,0),(s_1,1),(s_2,1),(s_3,0),(s_4,0),(s_5,2),(s_6,3),(s_7,1),(s_8,2)\}.$$

- *The* CR*-representation uses two BDDs* $\mathcal{P}^0_{\mathrm{CR}}(s)$ *and* $\mathcal{P}^1_{\mathrm{CR}}(s)$ *such that*

$$\mathcal{P}^0_{\mathrm{CR}}(s)=1 \quad \textit{iff} \quad s\in B_1\cup B_3=\{s_1,s_2,s_6,s_7\}$$
$$\mathcal{P}^1_{\mathrm{CR}}(s)=1 \quad \textit{iff} \quad s\in B_2\cup B_3=\{s_5,s_6,s_8\}.$$

We can access, for instance, block B_2 *by*

$$B_2=P^1_{\mathrm{CR}}\cap(S\setminus P^0_{\mathrm{CR}})=\{s_5,s_6,s_8\}\cap\{s_0,s_3,s_4,s_5,s_8\}=\{s_5,s_8\}.$$

The BDDs for both representation techniques are depicted in Figure 1. For the sake of readability, we have removed all edges which directly point to leaf 0.

These four representations differ in terms of their time and space efficiency when applied in a symbolic implementation of a refinement algorithm. We report on detailed experiments in Section 6, but provide a preview here for the sake of the exposition.

The size of the first representation, ER, is rather unpredictable: sometimes it is the most compact representation, in other cases it needs by far the most nodes of all four techniques. As shown in Section 6 by a number of example models, the second representation, BR, is not space efficient in practice either.

We classify FR as the technique that enables us to implement the key operations of the algorithm very efficiently in terms of running time. We therefore call it the *fast* representation. One disadvantage of the fast representation is its size: the size of the BDD $\mathcal{P}$ is at least linear in the number of blocks. More precisely, $|\mathcal{P}|=\Omega(|P|)$, and that makes it unsuitable for partitions with a large number of blocks. CR, in turn, is considerably slower to use, but in many cases has the smallest number of nodes among the four representations. Therefore, we call it the *compact* representation.

Quotient extraction from FR *and* CR*.* We briefly show how the quotient system with respect to a lumpable partition can be computed when the partition is represented using CR or FR. We describe the way how a symbolic representation of the quotient can be obtained. An explicit representation can be obtained from the symbolic representation by simple enumeration of all assignments that lead to a non-zero leaf of the (MT)BDDs. The pseudo-codes of both algorithms are given in Algorithm 4.1.

To extract the quotient from CR, one state is first selected from each block. These states form the state space $\widetilde{\mathcal{S}}$ of the quotient system (lines 2–5). The function GetBlock for accessing the blocks of a partition is

```
SymbolicCompQuotientCR($\mathcal{S}, \mathcal{R}, \mathcal{P}_{\mathrm{CR}}$)
1  $\widetilde{\mathcal{S}}(s) := 0$;  $\widetilde{\mathcal{R}}(s,t) := 0$;
2  for $c := 0$ to $\#\mathcal{P}_{\mathrm{CR}} - 1$
3     $\mathcal{B}_c(s) :=$ GetBlock($\mathcal{S}, \mathcal{P}_{\mathrm{CR}}, c$)
4     $\mathcal{X}_c(s) := \{$arbitrary element of $\mathcal{B}_c\}$
5     $\widetilde{\mathcal{S}}(s) := \widetilde{\mathcal{S}}(s) + \mathcal{X}_c(s)$
6  $\mathcal{R}'(s,t) := \mathcal{R}(s,t) \cdot \widetilde{\mathcal{S}}(s)$
7  for $c := 0$ to $\#\mathcal{P}_{\mathrm{CR}} - 1$
8     $\mathcal{R}''(s) := \Diamond_t^{+}\big(\mathcal{R}'(s,t) \cdot \mathcal{B}_c(t)\big)$
9     $\mathcal{R}''(s,t) := \mathcal{R}''(s) \cdot \mathcal{X}_c(t)$
10    $\widetilde{\mathcal{R}}(s,t) := \widetilde{\mathcal{R}}(s,t) + \mathcal{R}''(s,t)$
11 return $(\widetilde{\mathcal{S}}, \widetilde{\mathcal{R}})$
```

```
GetBlock($\mathcal{S}, \mathcal{P}_{\mathrm{CR}}, c$)
1  $\mathcal{B}(s) := \mathcal{S}(s)$
2  for $i := 0$ to $|\mathcal{P}_{\mathrm{CR}}| - 1$
3     if $i^{\mathrm{th}}$ bit of $c = 1$
4        $\mathcal{B}(s) := \mathcal{B}(s) \wedge \mathcal{P}^{i}_{\mathrm{CR}}(s)$
5     else
6        $\mathcal{B}(s) := \mathcal{B}(s) \wedge \neg\mathcal{P}^{i}_{\mathrm{CR}}(s)$
4  return $\mathcal{B}$
```

```
SymbolicCompQuotientFR($\mathcal{S}, \mathcal{R}, \mathcal{P}_{\mathrm{FR}}$)
1  $\widetilde{\mathcal{S}}(s) := [k \to s]\big(\exists s : \mathcal{P}_{\mathrm{CR}}(s,k)\big)$
2  $\mathcal{R}_1(s,t) := [k \to t](\Diamond_t^{+}\big(R(s,t) \cdot \mathcal{P}_{\mathrm{FR}}(t,k)\big))$
3  $\widetilde{\mathcal{R}}(s,t) := [k \to s](\Diamond_s^{\max}\big(\mathcal{R}_1(s,t) \cdot \mathcal{P}_{\mathrm{FR}}(s,k)\big))$
4  return $(\widetilde{\mathcal{S}}, \widetilde{\mathcal{R}})$
```

Algorithm 4.1: Symbolic quotient extraction for CR (top) and FR (bottom)

a direct implementation of formula (5). Then the transitions are restricted to the selected states (line 6). For each block $\mathcal{B}_c$ and state s, we compute the cumulative transition rate from s to $\mathcal{B}_c$ (line 8). Thereby, $\Diamond_x^{\odot}$ is the quantification operator with respect to an associative and commutative operator $\odot$:

$$\Diamond_x^{\odot}\big(\mathcal{A}(x,y)\big) = \bigodot_{a \in \{0,1\}^h} \mathcal{A}(a,y)$$

The selected state of the current block is added as the target state of the newly computed transitions (line 9). These are then added to the new transition matrix (line 10).

The method for FR proceeds in a different way: The block numbers are used as the states of the quotient system (line 1). The cumulative transition rates are computed in line 2, thereby substituting the target states by their block numbers. The operation $[k \to t](\cdot)$ renames the block number variables to the corresponding target state variables. And finally, the source states are replaced by their block numbers (line 3).

The resulting system is in both cases the quotient MC with respect to the lumpable partition $\mathcal{P}_{\mathrm{CR}}$ and $\mathcal{P}_{\mathrm{FR}}$, respectively. From now on, we will concentrate on the computation of such partitions.

4.2. Symbolic Version of the Traditional Algorithm

By replacing the explicit data structure by the compact MTBDD representation and the explicit operations by their symbolic counterparts, we can transform the explicit-state lumping algorithm from Section 3.1 to a symbolic one. This approach was used in [32].

The approach of the symbolic algorithm, whose pseudo-code is shown in Algorithm 4.2, is the same as for the explicit variant: as long as there are splitters, the blocks of the current partition are split successively. In the beginning the list of splitters is identical to the initial partition. Each time a block is split, all resulting blocks but one are added to the list of splitters and in the current partition, the block that was split is replaced by the parts into which it was split.

The difference is the administration of the list of splitters: To reduce the number of update operations, both the current partition and the current list of splitters are stored in $\mathcal{P}_{\mathrm{CR}}$. The next splitter is determined using a counter sc whose initial value is 0 and which is increased after each splitting round. When we split a block C into $C'_1, \ldots, C'_\alpha$, we replace C by C'_1 and append the other blocks $C'_2, \ldots, C'_\alpha$ to the end of the list. They will become splitters later when sc is increased. Hence, in the main loop of LumpablePartitionFR (lines 2–4), we iterate over all necessary splitters and refine the current partition with respect to them.

```
LumpablePartitionCR($\mathcal{P}_{\mathrm{CR}}^{(0)}$)
1  $\mathcal{P}_{\mathrm{CR}} := \mathcal{P}_{\mathrm{CR}}^{(0)}$
2  for $sc := 0$ to $\#\mathcal{P}_{\mathrm{CR}}$
3     $\mathcal{B} :=$ GetBlock($\mathcal{P}_{\mathrm{CR}}, sc$)
4     SymSplit($\mathcal{P}_{\mathrm{CR}}, \mathcal{B}$)
5  return $\mathcal{P}_{\mathrm{CR}}$
```

```
SymComputeKeys($\mathcal{R}, \mathcal{C}, \mathcal{B}$)
1  $\mathcal{R}'(s,t) := \mathcal{C}(s) \cdot \mathcal{B}(t) \cdot \mathcal{R}(s,t)$
2  $\mathcal{K} := \Diamond_t^{+}(\mathcal{R}'(s,t))$
3  return $\mathcal{K}$
```

```
SymSplit($\mathcal{P}_{\mathrm{CR}}, \mathcal{B}$)
1   for $c := 0$ to $\#\mathcal{P}_{\mathrm{CR}} - 1$
2      $\mathcal{C} :=$ GetBlock($\mathcal{P}_{\mathrm{CR}}, c$)
3      $\mathcal{K} :=$ SymComputeKeys($\mathcal{R}, \mathcal{C}, \mathcal{B}$)
4      $T := \{$leaves of $\mathcal{K}\}$
5      $\alpha := \#\mathcal{P}_{\mathrm{CR}} - 1$
6      foreach $x \in T$
7         $\mathcal{C}'_\alpha := (\mathcal{K} == x)$
8         $\alpha := \alpha + 1$
9      ReplaceBlock($\mathcal{P}_{\mathrm{CR}}, c, \mathcal{C}'_{\#\mathcal{P}_{\mathrm{CR}}-1}$)
10     for $i := \#\mathcal{P}_{\mathrm{CR}}$ to $\alpha - 1$
11        AddBlock($\mathcal{P}_{\mathrm{CR}}, i, \mathcal{C}'_i$)
```

```
AddBlock($\mathcal{P}_{\mathrm{CR}}, k, \mathcal{C}$)
1  if $k = 2^d$
2     Append $\emptyset$ to the end of $\mathcal{P}_{\mathrm{CR}}$
3     $d := d + 1$
4  for $i := 0$ to $d - 1$
5     if $i^{\mathrm{th}}$ bit of $k = 1$
6        $\mathcal{P}_{\mathrm{CR}}^i := \mathcal{P}_{\mathrm{CR}}^i \vee \mathcal{C}$
7  return $\mathcal{P}_{\mathrm{CR}}$
```

```
ReplaceBlock($\mathcal{P}_{\mathrm{CR}}, k, \mathcal{C}$)
1  $C_{\mathrm{old}} :=$ GetBlock($\mathcal{P}_{\mathrm{CR}}, k$)
2  for $i := 0$ to $d - 1$
3     if $i^{\mathrm{th}}$ bit of $k = 1$
4        $\mathcal{P}_{\mathrm{CR}}^i := (\mathcal{P}_{\mathrm{CR}}^i \wedge \neg C_{\mathrm{old}}) \vee \mathcal{C}$
5  return $\mathcal{P}_{\mathrm{CR}}$
```

Algorithm 4.2: The symbolic version of the traditional lumping algorithm

The splitting with respect to block $\mathcal{B}$ is done for each block $\mathcal{C}$ of the current partition by computing the key $\mathcal{K}(s) = \sum_{t \in B} R(s,t)$. The different rates into the splitter are stored in the leaves of $\mathcal{K}$. For each such value x, we determine the set of states with $\mathcal{K}(s) = x$ (line 7 of SymSplit). The first such block replaces $\mathcal{C}$ in $\mathcal{P}_{\mathrm{CR}}$, the other ones are appended to $\mathcal{P}_{\mathrm{CR}}$. This works as follows:

Assume $\mathcal{P}_{\mathrm{CR}} = (\mathcal{P}_{\mathrm{CR}}^0, \ldots, \mathcal{P}_{\mathrm{CR}}^{d-1})$. ReplaceBlock($\mathcal{P}_{\mathrm{CR}}, l, \mathcal{C}'_l$) replaces the block $\mathcal{C}_l$ with $\mathcal{C}'_l$. Therefore, $\mathcal{P}_{\mathrm{CR}}$ is updated in the following way:

$$\mathcal{P}_{\mathrm{CR}}^i = \begin{cases} \mathcal{P}_{\mathrm{CR}}^i & \text{if the } i^{\text{th}} \text{ bit of } l \text{ is zero} \\ (\mathcal{P}_{\mathrm{CR}}^i \wedge \neg C_l) \vee C'_l & \text{otherwise.} \end{cases}$$

The call AddBlock($\mathcal{P}_{\mathrm{CR}}, \mathcal{C}_k$) adds the block $\mathcal{C}_k$ at position k to $\mathcal{P}_{\mathrm{CR}}$. Before the call, GetBlock($\mathcal{P}_{\mathrm{CR}}, k$) must be empty. We have to update the partition in the following way:

$$\mathcal{P}_{\mathrm{CR}}^i = \begin{cases} \mathcal{P}_{\mathrm{CR}}^i & \text{if the } i^{\text{th}} \text{ bit of } k \text{ is zero} \\ \mathcal{P}_{\mathrm{CR}}^i \vee C_k & \text{otherwise.} \end{cases}$$

If $\mathcal{P}_{\mathrm{CR}} = (\mathcal{P}_{\mathrm{CR}}^0, \ldots, \mathcal{P}_{\mathrm{CR}}^{d-1})$ and $k = 2^d$, we add $\mathcal{P}_{\mathrm{CR}}^d = \emptyset$ to $\mathcal{P}_{\mathrm{CR}}$ before appending C_d.

Upon termination this yields the coarsest lumpable partition which refines $\mathcal{P}_{\mathrm{CR}}^{(0)}$. For more details, see [32].

4.3. Symbolic Implementation of the Signature-based Refinement Algorithm

Algorithm 4.3 shows the symbolic implementation of the signature-based algorithm explained in Section 3.2. Line 4 computes the MTBDD representation of the signatures, and SigRefine, in line 5, returns the partition refined with respect to the signatures. More details follow.

```
LumpablePartitionFR($\mathcal{P}_{\mathrm{FR}}^{(0)}$)
1  $\mathcal{P}'_{\mathrm{FR}} := \mathcal{P}_{\mathrm{FR}}^{(0)}$
2  repeat
3      $\mathcal{P}_{\mathrm{FR}} := \mathcal{P}'_{\mathrm{FR}}$
4      $\sigma(s,k) := \Diamond_t^+(\mathcal{R}(s,t) \cdot \mathcal{P}_{\mathrm{FR}}(t,k)) - \mathcal{D}(s) \cdot \mathcal{P}_{\mathrm{FR}}(s,k)$
5      nextNumber := 0
6      $\mathcal{P}'_{\mathrm{FR}}$ := SigRefine($\sigma$)
7  until ($\mathcal{P}_{\mathrm{FR}} = \mathcal{P}'_{\mathrm{FR}}$)
8  return $\mathcal{P}_{\mathrm{FR}}$

SigRefine($\sigma$)
1  if $\sigma \in$ ComputedTable
2      return ComputedTable[$\sigma$]
3  $x$ := topVar($\sigma$)
4  if $x$ is a state variable
5      $\mathcal{T}$ := SigRefine($\sigma_{|x=1}$)
6      $\mathcal{E}$ := SigRefine($\sigma_{|x=0}$)
7      $\mathcal{N}$ := CreateNode($x, \mathcal{T}, \mathcal{E}$)
8  else
9      $\mathcal{N}$ := Number2BDD(nextNumber)
10     nextNumber := nextNumber + 1
11 ComputedTable[$\sigma$] := $\mathcal{N}$
12 return $\mathcal{N}$
```

Algorithm 4.3: The state space lumping algorithm using signature-based refinement

Let $\mathcal{Q}$ be the MTBDD of the generator matrix and $\mathcal{P}_{\mathrm{FR}}$ be the fast representation of the current partition. We then compute an MTBDD representation $\sigma(s,k)$ of the signatures as follows:

We use an MTBDD $\sigma_P(s,k)$ to represent the signature of states in S with respect to a partition P that is defined in the following way: $\sigma_P(s,k) = r$ iff $(r, B_k) \in \mathsf{sig}(P,s)$. This MTBDD can be obtained using the equation

$$\sigma(s,k) = \Diamond_t^+\big(\mathcal{Q}(s,t) \cdot \mathcal{P}_{\mathrm{FR}}(t,k)\big). \tag{6}$$

The problem with Eq. (6) is that $|\mathcal{Q}|$ might be excessively large due to the non-zero diagonal elements. To tackle this problem, we rewrite Eq. (6):

$$\sigma(s,k) = \Diamond_t^+\big(\mathcal{R}(s,t) \cdot \mathcal{P}_{\mathrm{FR}}(t,k)\big) - \mathcal{D}(s) \cdot \mathcal{P}_{\mathrm{FR}}(s,k) \tag{7}$$

in which $\mathcal{D}(s) = \Diamond_t^+\big(\mathcal{R}(s,t)\big)$ and $\mathcal{R}$ is the MTBDD representation of the transition rate matrix. Remarkably, $|\mathcal{D}|$ often is very small.

To implement the refinement operation sigref symbolically, we exploit the following observation: Assume that we have a variable order in which all state variables precede the block number variables. Then, each state corresponds to a path in the MTBDD which ends in a node that represents the signature of this state. Furthermore, since we use reduced MTBDDs, the paths of all states with the same signature must lead to the same node. To obtain the refined partition, we simply replace the nodes representing signatures by new block numbers.

Function SigRefine implements the sigref operator in this way. It takes $\sigma(s,k)$ as input and returns the BDD of the refined partition in FR-based representation.

To ensure that every node which represents a signature is replaced with the same block number each time it is visited when traversing the MTBDD recursively, we store the node and its replacement in a hash map, called ComputedTable. At the beginning of the algorithm, in lines 1–2, we first check if we have visited (and replaced) the node before. If the current operand is contained in the ComputedTable, we return the corresponding result without re-computation.

Otherwise, the BDD representing the signatures is traversed recursively (lines 4–8) until a node is reached that represents a signature. The result is then the BDD of a new block number. Before returning the result, we create an entry in the ComputedTable. The runtime of this algorithm lies in $O(|\sigma|)$.

To take an arbitrary initial partition into account, we have two possibilities: either we refine only one block of the initial partition at a time [44] or we add an extra entry to each signature such that the signatures of states belonging to different blocks of the initial partition cannot be identical [33].

4.4. Discussion

The two different symbolic algorithms, that we have presented above, have different weaknesses and strengths. The first algorithm, which is based on the compact partition representation, is very efficient in terms of memory. But its drawback is that it needs access to single blocks for refinement whose extraction is expensive. On the other hand, the FR-based algorithm inherits the memory-inefficiency from its partition representation, but its great advantage is its runtime efficiency. There is no need to extract single blocks, but all blocks can be refined simultaneously in one step.

The goal we want to achieve in the next section is to combine the strengths of both algorithms into a single method that minimizes the necessary computation time, thereby not exceeding the available amount of memory.

5. Hybrid Representation

In Section 2.2, we presented four different partition representations, two of which have desirable properties: the compact representation (CR) is very efficient in terms of memory requirement, but its manipulation (such as adding and removing a block) is relatively expensive. On the other hand, the fast representation (FR) enables us to perform the operation sigref very efficiently in terms of speed, but its space requirement is high for partitions with a large number of blocks.

5.1. Overall Idea

To get the best of both representations, our conceptual innovation is to provide a "hybrid" representation of the partition that uses FR for computation and CR for storage. Recall that the core computational step, in each iteration, is computing the signature of *all states* with respect to the *current partition.* Our key idea is to use CR to represent the current partition, but to represent the subpartition of states for which the signature is computed in FR, and to convert that into CR after signature-based refinement. To enable this, we do not necessarily compute the signatures of *all* states simultaneously as in LumpablePartitionFR.

Instead, we do that in a number of steps. In each step, the signatures of all states in a *chunk* are computed. A chunk is a collection of blocks of the current partition. This leads to a time-space trade-off depending on the number of blocks we convert to FR for refinement. For the approach to be effective, it is important that the additional overhead due to conversion is low.

The pseudo-code of LumpablePartitionHybrid, the hybrid algorithm, is given in Algorithm 5.1(a). Its outer loop corresponds to the main loop of LumpablePartitionFR. In each iteration of the inner loop of LumpablePartitionHybrid, Chunk computes a chunk consisting of *csize* blocks of the current partition $\mathcal{P}_{\mathrm{CR}}$. Then, Signature computes the signatures of all states in the chunk. Based on the signatures, SigRefine refines the chunk into a subpartition in FR, and ConvertFR2CR converts the subpartition into compact representation. Finally, Union adds it to $\mathcal{P}'_{\mathrm{CR}}$, the compact representation of the (new) refined subpartition computed so far.

The hybrid version of the algorithm consumes less memory because it avoids representing both the signatures of all states and also the complete partition using FR (σ, $\mathcal{P}$, and $\mathcal{P}'$ in LumpablePartitionFR). Instead, at each point of time only one chunk ($\mathcal{C}_i$), the signatures of its states (σ_i), and its refinement represented as FR ($\mathcal{P}'_{\mathrm{FR},i}$) are stored. Since the number of nodes in the last two MTBDDs grows at least linearly with *csize*, we can adjust the memory consumption of the algorithm and achieve various time-space trade-offs by varying *csize*. That enables the hybrid algorithm to handle MCs out of reach of the original FR-based algorithm due to memory limitations.

```
(a) LumpablePartitionHybrid($\mathcal{P}_{\mathrm{CR}}^{(0)}, csize$)
$\mathcal{P}'_{\mathrm{CR}} := \mathcal{P}_{\mathrm{CR}}^{(0)}$
repeat
  $\mathcal{P}_{\mathrm{CR}} := \mathcal{P}'_{\mathrm{CR}}$; $\mathcal{P}'_{\mathrm{CR}} := \emptyset$; $\mathit{firstBlkIdx} := 0$
  for $i := 0$ to $\lceil \#\mathcal{P}_{\mathrm{CR}}/csize \rceil - 1$
    $\mathcal{C}_i := \textsc{Chunk}(\mathcal{P}_{\mathrm{CR}}, csize, i)$
    $\sigma_i(s,k) := \textsc{Signature}(\mathcal{C}_i)$
    $\mathcal{P}'_{\mathrm{FR},i} := \textsc{SigRefine}(\sigma_i, \mathit{firstBlkIdx})$
    $\mathcal{P}'_{\mathrm{CR},i} := \textsc{ConvertFR2CR}(\mathcal{P}'_{\mathrm{FR},i})$
    $\mathcal{P}'_{\mathrm{CR}} := \textsc{Union}(\mathcal{P}'_{\mathrm{CR}}, \mathcal{P}'_{\mathrm{CR},i})$
    $\mathit{firstBlkIdx} := \mathit{firstBlkIdx} + \#\mathcal{P}'_{\mathrm{FR},i}$
until $(\#\mathcal{P}_{\mathrm{CR}} = \#\mathcal{P}'_{\mathrm{CR}})$
return $\mathcal{P}_{\mathrm{CR}}$
```

```
(b) Chunk($\mathcal{P}_{\mathrm{CR}}, csize, i$)
$\mathcal{C} := S$
if $\#\mathcal{P}_{\mathrm{CR}} > csize$
  $cbits := \log_2 csize$
  for $j := 0$ to $(d-1) - cbits$
    if $j^{\mathrm{th}}$bit of $i = 1$
      $\mathcal{C} := \mathcal{C} \cap \mathcal{P}_{\mathrm{CR}}^{j+cbits}$
    else
      $\mathcal{C} := \mathcal{C} \cap (S \setminus \mathcal{P}_{\mathrm{CR}}^{j+cbits})$
    end if
end if
return $\mathcal{C}$
```

```
(c) Signature($\mathcal{C}_i$)
$\mathcal{C}'_i(s) := [t \to s] \left(\lozenge_s^{\vee}(\mathcal{T}(s,t) \cdot \mathcal{C}_i(s))\right)$
$\mathcal{A}(s,k) := \textsc{BlockToFR}(\mathcal{C}'_i(s))$
$\mathcal{R}'(s,t,k) := (\mathcal{R}(s,t) \cdot \mathcal{C}_i(s)) \cdot \mathcal{A}(t,k)$
$\mathcal{D}'(s,k) := \mathcal{D}(s) \cdot \mathcal{C}_i(s) \cdot \mathcal{A}(s,k)$
$\sigma_i(s,k) := \lozenge_t^{+}(\mathcal{R}'(s,t,k)) - \mathcal{D}'(s)$
return $\sigma_i$
```

Algorithm 5.1: State space lumping algorithm using hybrid partition representation

The extraction of the quotient system after computing a lumpable partition can be performed exactly as described in Algorithm 4.1 (upper part), since the final partition is given in CR.

5.2. Signature Refinement in the Hybrid Representation

In the following, we will explain in detail how each of the steps of LumpablePartitionHybrid is performed.

Computing the chunks. As mentioned above, LumpablePartitionHybrid partitions the state space into chunks and then it refines each chunk separately. The function Chunk($\mathcal{P}_{\mathrm{CR}}, csize, i$) computes the i^{th} chunk consisting of *csize* blocks of P represented compactly as $\mathcal{P}_{\mathrm{CR}}$. In other words, it returns the union of blocks of P with indices $i \cdot csize$ through $\min\{(i+1) \cdot csize, |P|\} - 1$.

Algorithm 5.1(b) shows the pseudo-code of Chunk. The variable d is the number of MTBDDs of the compact representation of P. We restrict *csize* to be a power of 2, which enables us to perform Chunk using $O(\log |P|)$ symbolic operations.

Conversion from FR *to* CR. Let $\mathcal{P}_{\mathrm{FR}}(s,k)$ be the fast representation of the partition $P = \{B_1, \ldots, B_n\}$ and $(\mathcal{P}_{\mathrm{CR}}^0, \ldots, \mathcal{P}_{\mathrm{CR}}^{d-1})$ be its compact representation. Recall that $\mathcal{P}_{\mathrm{CR}}^j$ is the union of all blocks B_i such that the j^{th} bit of i is one. Consequently, $S \setminus \mathcal{P}_{\mathrm{CR}}^j$ is the union of all blocks B_i such that the j^{th} bit of i is zero.

Therefore, to convert $\mathcal{P}_{\mathrm{FR}}$ to the compact representation, we have

$$\mathcal{P}_{\mathrm{CR}}^j(s) = \lozenge_k^{\vee}\left(\mathcal{P}_{\mathrm{FR}}(s,k)_{|k_j=1}\right). \qquad (8)$$

Observe that the conversion takes only $O(d) = O(\log |P|)$ symbolic operations. Notably, the conversion from CR to FR can also be done in a logarithmic number of steps, but is not needed for the algorithm.

Computing the signatures. Signature($\mathcal{C}_i$) (see Algorithm 5.1(c)) computes the signatures of all states in $\mathcal{C}_i$, the i^{th} chunk of the current partition $\mathcal{P}_{\mathrm{CR}}$. In Signature, $\mathcal{T}$ is the MTBDD of the 0-1 transition matrix, i. e., $\mathcal{T}(s,t) = 1$ if there is a transition from s to t, and $\mathcal{T}(s,t) = 0$ otherwise. We first compute $\mathcal{C}'_i$ the set of successor states of $\mathcal{C}_i$. Then, BlockToFR attaches to each state in $\mathcal{C}_i$ its block number and stores it in $\mathcal{A}$, i. e., $\mathcal{A}(s,k) = 1$ iff $s \in B_k \cap \mathcal{C}'_i$, and $\mathcal{A}(s,k) = 0$ otherwise. We explain below how this is performed efficiently. The remaining three lines provide the restriction of the source states in the generator matrix $\mathcal{Q}(s,t)$ to the current chunk $\mathcal{C}_i$, and replace the target states therein with their block numbers. As discussed in Section 4.3 for Eq. (7), we avoid the representation of the generator matrix in this computation.

The central novelty in SIGNATURE is BLOCKTOFR($\mathcal{U}$). It computes the fast representation of the restriction of a partition on the set $U \subseteq S$. The straightforward approach to do so is to compute $\mathcal{P}_{\mathrm{FR}}(s,k)\cdot\mathcal{U}(s)$. However, that necessitates the generation of the fast (and possibly large) representation $\mathcal{P}_{\mathrm{FR}}$ of P which we want to avoid. BLOCKTOFR exploits the compact representation and performs only $O(d) = O(\log|P|)$ symbolic operations to do so:

$$\textsc{BlockToFR}(\mathcal{U}) = \bigwedge_{j=0}^{d-1} \Big(k_j \wedge (\mathcal{P}^j_{\mathrm{CR}} \cap \mathcal{U})\Big) \vee \Big(\overline{k_j} \wedge \Big((S \setminus \mathcal{P}^j_{\mathrm{CR}}) \cap \mathcal{U}\Big)\Big) \tag{9}$$

The intuition behind Eq. (9) is that $k_j \wedge (\mathcal{P}^j_{\mathrm{CR}} \cap \mathcal{U})$ (resp. $\overline{k_j} \wedge ((S \setminus \mathcal{P}^j_{\mathrm{CR}}) \cap \mathcal{U})$) sets to 1 (resp. 0) the j^{th} block index variable of all states in U that belong to a block whose index has 1 (resp. 0) in its j^{th} bit.

Computing the union of two partitions. In LUMPABLEPARTITIONHYBRID, we refine a chunk using the signatures of its states resulting in $\mathcal{P}'_{\mathrm{CR},i}$, a partition of the chunk. Then, UNION computes a new subpartition that contains all blocks of $\mathcal{P}'_{\mathrm{CR},i}$ and $\mathcal{P}'_{\mathrm{CR}}$, the computed subpartition of the state space. In the new subpartition, all the blocks have the same index as they had in $\mathcal{P}'_{\mathrm{CR}}$ or $\mathcal{P}'_{\mathrm{CR},i}$.

Lemma 1 forms the basis of the UNION operation:

Lemma 1. *Let P be a partition of $U \subseteq S$ and $\{P', P''\}$ be a partition of P. Furthermore, let $(P_0, \ldots, P_{d-1})$, $(P'_0, \ldots, P'_{d-1})$, and $(P''_0, \ldots, P''_{d-1})$ be the* CR *of P, P', and P'' respectively. Then,*

$$P_j = P'_j \cup P''_j. \tag{10}$$

PROOF.

$$\begin{aligned} P_j &= \bigcup\{B_i \in P \mid \text{the } j^{\mathrm{th}} \text{ bit of } i \text{ is } 1\} \\ &= \bigcup\{B_i \in P' \mid \text{the } j^{\mathrm{th}} \text{ bit of } i \text{ is } 1\} \cup \bigcup\{B_i \in P'' \mid \text{the } j^{\mathrm{th}} \text{ bit of } i \text{ is } 1\} \\ &= P'_j \cup P''_j \end{aligned}$$

□

If the CR of P' and P'' have different lengths, we append empty sets to the shorter representation to bring them to the same length. Using Eq. (10), UNION(P', P'') takes only $\log_2(|P'| + |P''|)$ symbolic union operations.

To make Lemma 1 applicable and perform UNION($\mathcal{P}'_{\mathrm{CR}}, \mathcal{P}'_{\mathrm{CR},i}$) efficiently, we need to assign the block numbers carefully. In particular, we need the indices of $\mathcal{P}'_{\mathrm{CR}}$ and $\mathcal{P}'_{\mathrm{CR},i}$ to be disjoint and also not to have "holes" (i. e., to have consecutive block indices). Hence, we only have to compute the union of the corresponding BDDs. To facilitate this, we always make the indices of blocks of $\mathcal{P}'_{\mathrm{CR}}$ start from zero in each iteration of the outer loop. When we want to index the blocks of $\mathcal{P}'_{\mathrm{FR},i}$, we set our starting index to be $\#\mathcal{P}'_{\mathrm{CR}}$, i. e., the next available index in $\mathcal{P}'_{\mathrm{CR}}$. That is reflected in the second parameter of SIGREFINE in Algorithm 5.1(a).

5.3. Optimizations

Avoiding unnecessary conversions. During the initial iterations of the outer loop of LUMPABLEPARTITION-HYBRID, the number of blocks of P is normally smaller than the chunk size *csize*. That means the whole partition consists only of one chunk. In that case, we completely avoid the overhead of the conversions by only using FR until the number of blocks exceeds *csize*. Only then do we start the refinement with the hybrid partition representation.

As a result, the algorithm automatically switches the partition representation from FR to hybrid only when it is necessary. The advantage is that the algorithm achieves the efficiency of the algorithm in [33] (solely based on FR) as long as the resulting partition does not exceed *csize*.

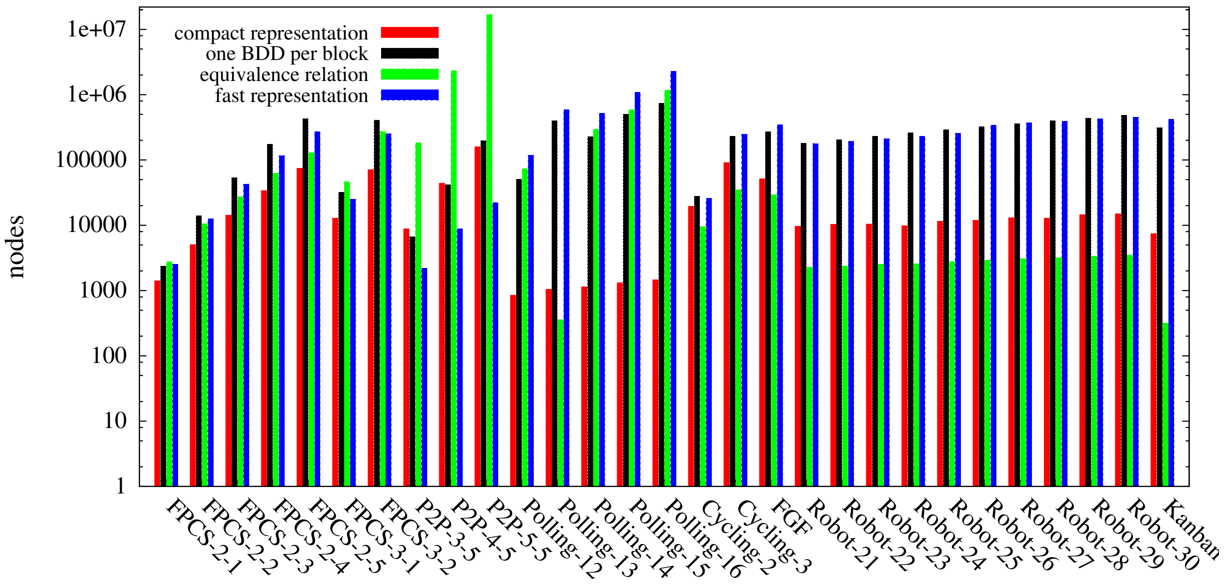


Figure 2: Sizes of the partitions using various representation techniques. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article)

Automatic selection of csize. As explained above, *csize* enables us to change the time–space balance of the algorithm. As we increase *csize*, the running time of the algorithm decreases and its space consumption increases. Therefore, it is important to be able to automatically choose an appropriate value for *csize*. In the following, we present an algorithm to do so.

Our goal is to use as much of the available physical memory as possible, thereby minimizing the computation time. There are ample possibilities to deal with the time–space trade-off behind the chunk size. We implemented the following.

We let the user provide a memory limit. Initially, we set $csize = 2^{\lceil \log_2 |S| \rceil}$ (upper bound) such that all blocks fit into one chunk. Therefore, the algorithm starts using only FR. If the memory limit is exceeded, we set $csize = 2^{\lceil \log_2 |P| \rceil - 1}$ where P is the last successfully computed partition. The algorithm is then restarted using P as the initial partition. Since $csize < |P|$, the algorithm uses a hybrid of CR and FR, thereby reducing the memory consumption at the price of higher runtime. Afterward, each time the memory limit is exceeded, the chunk size is divided by two and the algorithm is restarted using the last successfully computed partition as the initial partition.

If the chunk size becomes smaller than one, the refinement process cannot be completed within the given memory bound. Either the algorithm has to be aborted or the memory limit has to be increased.

6. Experimental Study

6.1. Example Models and Implementation

We have implemented our hybrid algorithm in C++ using the CUDD package [45] as the MTBDD library. To generate the MTBDD representations of the input MCs, we used the probabilistic model checking tool PRISM [21]. Our input models are given in PRISM's guarded command language. They are read by PRISM, which generates MTBDD representations of the models. We have modified PRISM such that the MTBDD representations are dumped to a file. These dumped MTBDDs are read by our implementation.

All the code involved in the experiments was compiled using gcc 4.2.4. The experiments were conducted on a Dual Core AMD Opteron$^{\mathrm{TM}}$ 2.4 GHz CPU with 4 GB of main memory running Linux in 32-bit mode. We have stopped any experiment that takes more than 2000 seconds. This value is chosen arbitrarily, but uniform over all experiments.

Table 1: Size of the example models before and after lumping

	Original System		Quotient System	
Benchmark	states	transitions	states	transitions
FPCS-2-2	15769	91232	703	5846
FPCS-2-3	256932	1697760	2145	19760
FPCS-2-4	3803193	27771984	5151	50298
FPCS-2-5	52639632	416078208	10585	107300
FPCS-3-1	23040	153600	969	8908
FPCS-3-2	1889947	15302784	9139	107522
FPCS-3-3	124075800	1151306784	47905	631280
P2P-3-5	32768	245761	56	106
P2P-4-5	1048576	10485761	126	281
P2P-5-5	33554432	419430401	196	456
P2P-6-5	1073741824	16106127361	266	631
Polling-12	73728	503808	6144	41984
Polling-13	159744	1171456	12288	90112
Polling-14	344064	2695168	24576	192512
Polling-15	737280	6144000	49125	409600
Polling-16	1572864	13893632	98304	868352
Polling-18	7077888	69599232	393216	3866624
Cycling-2	4666	18342	3511	14445
Cycling-3	57667	305502	40659	224591
Cycling-4	431101	2742012	282943	1878339
Cycling-5	2326666	16778785	1424914	10739110
FGF	80616	562536	38639	379757
Robot-25	61200	325917	60600	322919
Robot-26	68900	367397	68250	364149
Robot-27	77220	412253	76518	408745
Robot-28	86184	460617	85428	456839
Robot-29	95816	512621	95004	508563
Robot-30	106140	568397	105270	564049
Kanban-3	58400	446400	58400	446400

We consider seven different example models from the literature to study the performance of the algorithm: A fault-tolerant parallel computer system (`FPCS`) [46], a peer-to-peer (`P2P`) protocol based on BitTorrent (studied in [8]), a cyclic server polling system [47], a robot moving through an $n \times n$ grid [48] (`Robot`), a Kanban production system [49], and two biological models: the first one describes the Fibroblast growth factor signaling (`FGF`) within cells [50], and the second one is a probabilistic model of cell cycle control in eukaryotes (`Cycling`) [51].

For the FPCS model, we converted the SAN (Stochastic Activity Network) specification to the PRISM input language. We obtained the PRISM specifications of the other five models from `http://www.prismmodelchecker.org/casestudies/index.php`.

All but the FGF model are parametrized. The first two models have two parameters. For FPCS, they denote the number of computers in the system and the number of memory modules in each computer, respectively. For `P2P`, they represent the number of clients and the number of blocks of the file to be transmitted, respectively. The remaining models have only one parameter: for the polling benchmark, the parameter denotes the number of servers; for the robot benchmark, the size of the grid; in the Kanban benchmark, the parameter denotes the number of tokens in the system; and for the cell cycle control, it denotes the initial number of molecules.

With the exception of the Kanban model, all of these Markov chains can be minimized, i. e., the lumped system is smaller than the original one. The lumped model of the Kanban system, however, has the same size as the input model.

Table 1 shows the number of states and transitions for all example models before and after lumping.

The runtimes for all following experiments include the time to compute the coarsest ordinarily lumpable partition, but not the time for computing the quotient system using that partition. The reason is that the computation of the partition is the most expensive step, which is optimized by the hybrid algorithm. Using the hybrid algorithm with unlimited memory and the CR-based quotient extraction algorithm yielding a symbolic representation of the quotient MC, the quotient computation took less than 20 % of the overall minimization time.

6.2. Results

Partition sizes. We first computed the coarsest lumpable partition for the example models and converted them to the four partition representations described in Section 2.2 to compare their sizes.

For the representation of the equivalence relation we used an interleaved variable order, since an interleaved variable order often leads to small (MT)BDDs [52]. For the fast representation we used a variable order such that the block number variables are placed at the end of the order (i. e. at the bottom of the BDD). We need such a variable order to be able to perform the refinement operation efficiently (see Section 4.3).

Figure 2 shows the result of this experiment. Note that the scale of the vertical axis is logarithmic.

For all benchmarks with the exception of `P2P`, the CR representation (shown using the left-most (red) bars of each benchmark) ranks among the two most compact representations, often with only a small difference to the most compact one, while the other representations are, in some cases, significantly larger. Quantitatively speaking, the size of CR is on average[1] 2.24 times the size of the smallest representation while the same number for FR, BR and ER is 26.02, 28.87, and 5.03, respectively.

The exception is `P2P` for which CR is slightly, but not prohibitively, larger than FR. The reason is that `P2P` models exhibit a large degree of symmetry, resulting in quite small lumpable partitions (in the order of a few hundred blocks, see above). In this case, the overhead introduced by the block number variables in FR is marginal and the MTBDD size is dominated by other effects.

Effect of the chunk size. To evaluate the effect of the chunk size on the time and space requirements of the hybrid algorithm from Section 5, we applied it to the six models, varying the chunk size from one block per chunk to a size such that all blocks fit into one chunk. The maximal number of BDD nodes which have to be stored in memory simultaneously (peak number of nodes) is depicted on the left-hand side of Figure 3 and Figure 4 while the runtime is shown on the right-hand side.

We observe that for all benchmarks with the exception of `P2P`, the memory requirement grows drastically as the chunk size increases. For `P2P`, the memory requirement slightly decreases or stays more or less constant depending on the configuration. That is because the compact representation for the `P2P` model accounts for the major part of the memory requirement of the algorithm, as reflected in Figure 2.

Moreover, the runtime of the algorithm for all benchmarks decreases significantly when chunk size increases. The sensitivity of the runtime is milder for `P2P` examples because the chunk size has only little influence on the size of the partition representation.

Because of the optimization explained in Section 5.3, when *csize* is larger than the size of the lumpable partition (i. e., the final result), the hybrid algorithm behaves exactly like the pure-FR algorithm (that only uses FR). That is the reason why the runtime and space requirement of the hybrid algorithm eventually stabilizes when the chunk size exceeds a specific threshold.

In general, the experiments show that we can indeed have a time–space trade-off. The smaller the chunk size, the less memory and the more time the algorithm consumes. That property enables us to have a simple and automatic chunk size selection algorithm, described in Section 5.3.

Evaluation of the automatic chunk size selection algorithm. We ran the algorithm once without limiting the available memory, i. e., it was allowed to use as much physical memory as needed. In the other experiments, we limited the available memory to 75, 50, 30, and 15 MB for the MTBDDs. The results are shown in Table 2.

For each set of experiments, we give the running time, the memory usage of the nodes, and the number of chunk bits ($cbits = \log_2 csize$) with which the lumping computation succeeded without violating the memory limit[2]. We have marked the number of chunk bits using a bold font if it was automatically decreased by the

[1] This is computed using arithmetic mean.

[2] The actual memory requirement is slightly higher than the limit, since the limit is placed on the memory used by the MTBDDs. The memory requirement of the program code and of other data structures than MTBDDs is not taken into account for the limit.

Table 2: Runtime and memory consumption of the hybrid algorithm with automatic chunk size selection for different memory limitations (all times in seconds, memory usage in MB)

	unlimited memory			75 MB			50 MB		
Model	Time	Mem.	cbits	Time	Mem.	cbits	Time	Mem.	cbits
FPCS-2-2	0.32	18.29	15	0.34	18.29	15	0.30	18.29	15
FPCS-2-3	1.62	50.02	19	1.56	50.02	19	1.54	50.02	19
FPCS-2-4	6.51	62.93	23	6.32	62.94	23	6.66	55.93	23
FPCS-2-5	44.83	96.32	27	149.03	79.66	**13**	40.81	65.00	**12**
FPCS-3-1	0.80	29.45	16	0.84	29.46	16	0.84	29.45	16
FPCS-3-2	19.78	105.66	22	64.41	80.08	**12**	28.57	63.08	**12**
FPCS-3-3	220.00	520.70	28	329.63	89.27	**10**	296.09	61.61	**8**
P2P-3-5	0.07	10.49	15	0.07	10.49	15	0.08	10.49	15
P2P-4-5	0.92	33.55	20	0.90	33.55	20	0.91	33.55	20
P2P-5-5	6.94	63.68	25	6.99	63.67	25	6.22	54.65	25
P2P-6-5	227.86	62.72	30	230.57	62.72	30	152.97	56.51	30
Polling-12	24.92	60.07	17	23.76	60.08	17	23.53	56.85	17
Polling-13	71.89	71.99	18	72.01	71.99	18	71.99	68.18	18
Polling-14	260.64	110.79	19	391.42	87.13	**14**	350.88	68.61	**13**
Polling-15	710.72	231.48	20	789.63	85.47	**13**	802.45	66.48	**12**
Polling-16	1868.24	433.97	21	1965.46	83.67	**12**	2525.20	68.51	**11**
Polling-18	15610.48	1814.29	24		mem out			mem out	
Cycling-2	1.04	23.70	14	1.02	23.70	14	1.04	23.70	14
Cycling-3	26.05	88.55	17	27.10	83.53	17	38.26	64.62	**14**
Cycling-4	329.06	634.85	20	2369.78	87.51	**8**		mem out	
Cycling-5	3236.05	2132.23	23		mem out			mem out	
FGF	94.67	128.22	18	179.51	80.64	**13**	108.63	64.37	**13**
Robot-25	115.70	76.27	17	115.40	76.27	17	117.25	66.98	17
Robot-26	154.37	81.50	18	153.77	81.50	18	165.68	67.61	18
Robot-27	181.25	83.07	18	180.03	83.07	18	290.24	66.78	**16**
Robot-28	208.68	87.01	18	206.53	84.34	18	298.31	66.71	**16**
Robot-29	245.27	88.13	18	246.25	85.31	18	335.13	64.53	**15**
Robot-30	287.26	93.45	18	292.23	85.96	18	391.78	64.66	**15**
Kanban-3	53.24	146.71	17	69.22	82.55	**14**	67.81	67.91	**14**

	30 MB			15 MB		
Model	Time	Mem.	cbits	Time	Mem.	cbits
FPCS-2-2	0.34	18.29	15	0.34	18.29	15
FPCS-2-3	1.62	38.64	19	19.41	19.40	**11**
FPCS-2-4	34.66	37.20	**11**	19.96	20.45	**9**
FPCS-2-5	102.74	41.56	**10**		mem out	
FPCS-3-1	0.78	29.46	16	0.94	19.76	16
FPCS-3-2	47.06	40.41	**9**	30.94	25.05	**8**
FPCS-3-3	445.03	51.19	**7**		mem out	
P2P-3-5	0.07	10.49	15	0.08	10.49	15
P2P-4-5	0.93	33.55	20	1.16	19.41	20
P2P-5-5	25.62	35.26	25		mem out	
P2P-6-5		mem out			mem out	
Polling-12	24.74	40.02	17	49.63	20.73	**11**
Polling-13	144.36	43.46	**13**	130.03	25.11	**10**
Polling-14	297.24	50.51	**12**	466.23	24.95	**9**
Polling-15	826.32	48.64	**11**		mem out	
Polling-16	3566.04	46.14	**8**		mem out	
Polling-18		mem out			mem out	
Cycling-2	1.12	23.70	14	1.24	19.85	14
Cycling-3	71.88	39.33	**10**	179.33	20.96	**7**
Cycling-4		mem out			mem out	
Cycling-5		mem out			mem out	
FGF	113.76	40.54	**12**	139.06	22.84	**10**
Robot-25	178.65	39.96	**15**	201.39	21.02	**13**
Robot-26	239.90	38.42	**14**	264.93	20.85	**13**
Robot-27	279.88	37.19	**14**	324.10	20.97	**13**
Robot-28	342.43	38.39	**14**	353.87	21.01	**13**
Robot-29	368.46	37.57	**14**	417.22	20.85	**13**
Robot-30	416.45	38.70	**14**	476.56	20.57	**13**
Kanban-3	78.01	36.75	**12**		mem out	

Table 3: Comparison of the hybrid algorithm with the algorithms based only on FR [33] and on CR [32] (runtimes measured in seconds).

	pure FR-based algorithm		hybrid algorithm with automatic *csize*		pure CR-based algorithm	
Model	Runtime	Node peak	Runtime	Node peak	Runtime	Node peak
FPCS-2-2	0.36	192342	0.34	194100	2.43	30114
FPCS-2-3	1.56	646403	1.56	649118	27.31	56076
FPCS-2-4	6.24	1740601	6.32	1744309	148.07	153937
FPCS-2-5	mem out		149.03	3479910	669.45	365859
FPCS-3-1	0.82	429480	0.84	433907	9.41	74562
FPCS-3-2	mem out		64.41	3471734	785.16	501825
FPCS-3-3	mem out		329.63	3499328	time out	
P2P-3-5	0.07	35439	0.07	35548	0.95	39144
P2P-4-5	0.93	173396	0.90	173545	11.46	223080
P2P-5-5	6.84	502239	6.99	502428	88.75	684236
P2P-6-5	226.79	1064528	230.57	1064757	320.89	1554417
Polling-12	23.51	853853	23.76	854632	175.13	129042
Polling-13	71.90	1801559	72.01	1802466	545.42	466033
Polling-14	mem out		391.42	3507504	1788.61	452243
Polling-15	mem out		789.63	3483998	time out	
Polling-16	mem out		1965.46	3462536	time out	
Cycling-2	1.04	253277	1.02	257299	23.30	70223
Cycling-3	27.04	3283756	27.10	3295680	time out	
Cycling-4	mem out		time out		time out	
Cycling-5	mem out		mem out		mem out	
FGF	mem out		179.51	3503416	1098.53	390501
Robot-25	133.93	1905411	115.40	1906895	1312.20	246678
Robot-26	154.98	2339002	153.77	2340408	1642.80	265755
Robot-27	214.09	2556052	180.03	2557574	time out	
Robot-28	230.30	2770960	206.53	2772461	time out	
Robot-29	252.71	3047872	246.25	3049540	time out	
Robot-30	304.78	3294827	292.23	3296445	time out	
Kanban-3	mem out		69.22	3493196	time out	

chunk size selection algorithm to adhere to the memory limit. An entry "mem out" means that the lumping computation failed since the memory limit was exceeded in spite of a chunk size of 1.

We observe that as we decrease the physical memory available to the algorithm, it adapts itself by decreasing the chunk size as much as necessary. This adaptation comes with a cost in the running time which in the worst case grows by a small factor.

One would expect that the running time grows as the available memory decreases in each row of the table. However, there are a few exceptions. This is due to the caching behavior of the MTBDD package. If CUDD runs short of available memory, garbage collection is executed more frequently to free nodes which are no longer used. Furthermore, the size of internal caches is reduced to save memory. Both can influence the runtime in unpredictable ways. These memory saving strategies of CUDD are also the reason why sometimes less memory is used although the chunk size has not been decreased (see e. g., `Polling-12`).

Comparing the hybrid algorithm with the pure CR*-based algorithm (Section 4.2) and the pure* FR*-based algorithm (Section 4.3).* To have a fair comparison of the effectiveness of the three algorithms, we apply them to different models while we set an equal memory limit (75 MB) and time limit (2000 s) for all of them.

Table 3 shows the results of our experiments. For the hybrid algorithm the table contains the runtime including the time it takes to arrive at a suitable chunk size. We observe that the hybrid algorithm finishes several experiments successfully for which the FR-based and the CR-based algorithms fail due to memory limits and time limits, respectively. In all experiments that both hybrid and FR-based algorithms finish successfully, the hybrid algorithm is at most 2 % slower and uses roughly the same amount of memory. Moreover, for all experiments that both hybrid and CR-based algorithms finish successfully, the hybrid algorithm is much faster than the CR-based one.

If we relax the memory bound and provide 2 GB of main memory, the hybrid algorithm is the only one that does not fail on the `Cycling-5` benchmark. The purely FR-based algorithm fails due to the memory limit, whereas the CR-based one does not terminate within 12 hours.

Thus, our hybrid algorithm provides us with the core advantages of both representations/algorithms with a negligible running time penalty.

7. Conclusion

In this paper, we have developed a general, fast and memory efficient algorithm for ordinary (and also exact and strict) Markov chain lumping. The algorithm is presented in the context of continuous-time Markov chains, but is easily adaptable to labeled transition systems, Kripke structures, discrete-time or interactive Markov chains, Markov reward models, etc.

The particular strength of this algorithm is that it exploits the true potential of BDD-based representation with respect to time *and* space, in a way that so far was unavailable. Based on experimental analysis of different partition representation techniques, we have devised an algorithm that (1) exploits the compactness of the CR representation and (2) uses the efficiency of the FR representation for an iterative signature-based refinement of partition chunks. The algorithm is parametric in the chunk size it processes at once. We have also devised and evaluated a strategy that automatically chooses an appropriate value for the chunk size. Thanks to our hybrid representation and automatic chunk size selection, severe memory limitations caused only a worst case slowdown by a small factor in running time compared to unlimited available memory. Moreover, given the same memory limits, the hybrid algorithm works virtually as fast as pure-FR algorithm while drastically outperforming the pure-CR algorithm for models for which pure-FR runs out of memory.

The algorithm is designed to work with MTBDDs or BDDs. It appears possible to extend the algorithm to EVBDDs or PDGs, but it is unclear whether this has positive effects on the overall performance, since a key point is the partition representation. The number of inner nodes of a BDD and of its corresponding EVBDD is the same. Only the edges to the leaf 1 of the BDD are replaced by edges to the leaf 0 (with appropriate labeling) [18]. Therefore, using EVBDDs instead of BDDs for the partitions is not expected to reduce their size notably.

References

[1] D. Park, Concurrency and automata on infinite sequences, in: P. Deussen (Ed.), Proc. of the 5th GI-Conference on Theoretical Computer Science, Vol. 104 of Lecture Notes in Computer Science, Springer, 1981, pp. 167–183.

[2] J. G. Kemeney, J. L. Snell, Finite Markov Chains, D. Van Nostrand Company, Inc., 1960.

[3] W. H. Sanders, J. F. Meyer, Reduced base model construction methods for stochastic activity networks, IEEE Journal on Selected Areas in Communication 9 (1) (1991) 25–36.

[4] H. Hermanns, M. Ribaudo, Exploiting symmetries in stochastic process algebras, in: 12th European Simulation Multiconference, 1998, pp. 763–770.

[5] S. Gilmore, J. Hillston, M. Ribaudo, An efficient algorithm for aggregating PEPA models, IEEE Transactions on Software Engineering 27 (5) (2001) 449–464.

[6] G. Chiola, C. Dutheillet, G. Franceschinis, S. Haddad, Stochastic well-formed colored nets and symmetric modeling applications, IEEE Transactions on Computers 42 (11) (1993) 1343–1360.

[7] S. Derisavi, P. Kemper, W. H. Sanders, Symbolic state-space exploration and numerical analysis of state-sharing composed models, Linear Algebra and Its Applications 386 (2004) 137–166.

[8] M. Kwiatkowska, G. Norman, D. Parker, Symmetry reduction for probabilistic model checking, in: Proc. of the Int'l Conf. on Computer-Aided Verification (CAV), Vol. 4114 of Lecture Notes in Computer Science, 2006, pp. 234–248.

[9] K. G. Larsen, A. Skou, Bisimulation through probabilistic testing, in: 16th Annual ACM Symposium on Principles of Programming Languages (POPL), 1989, pp. 344–352.

[10] J. Hillston, A Compositional Approach to Performance Modelling, Cambridge University Press, 1996.

[11] J. Hillston, H. Hermanns, U. Herzog, V. Mertsiotakis, M. Rettelbach, Stochastic process algebras: integrating qualitative and quantitative modelling, in: 7th IFIP WG6.1 International Conference on Formal Description Techniques (FORTE), Vol. 6 of IFIP Conference Proceedings, Chapman & Hall, 1994, pp. 449–451.

[12] P. Buchholz, Exact and ordinary lumpability in finite Markov chains, Journal of Applied Probability 31 (1994) 59–74.

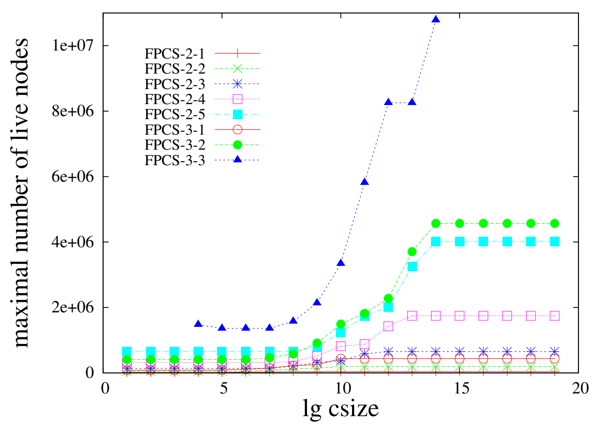
[13] S. Derisavi, H. Hermanns, W. H. Sanders, Optimal state-space lumping in Markov chains, Information Processing Letters 87 (6) (2003) 309–315.

[14] J.-P. Katoen, T. Kemna, I. Zapreev, D. N. Jansen, Bisimulation minimization mostly speeds up probabilistic model checking, in: Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Anlysis of Systems (TACAS), Vol. 4424 of Lecture Notes in Computer Science, 2007, pp. 87–101.

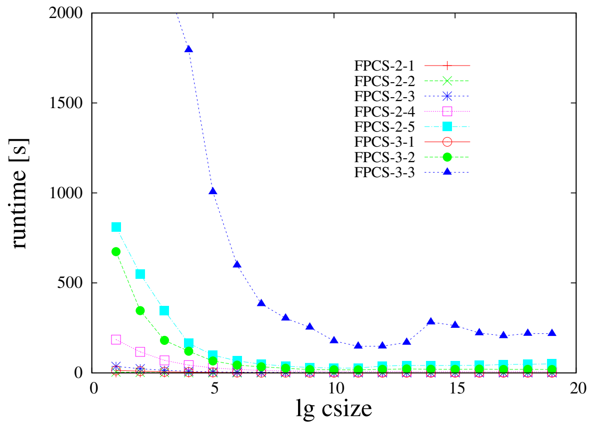
[15] E. Clarke, M. Fujita, P. McGeer, K. McMillan, J. Yang, X. Zhao, Multiterminal binary decision diagrams: An efficient data structure for matrix representation, Formal Methods in System Design 10 (2/3) (1997) 149–169.

[16] I. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, F. Somenzi, Algebraic decision diagrams and their applications, Formal Methods in System Design 10 (2/3) (1997) 171–206.

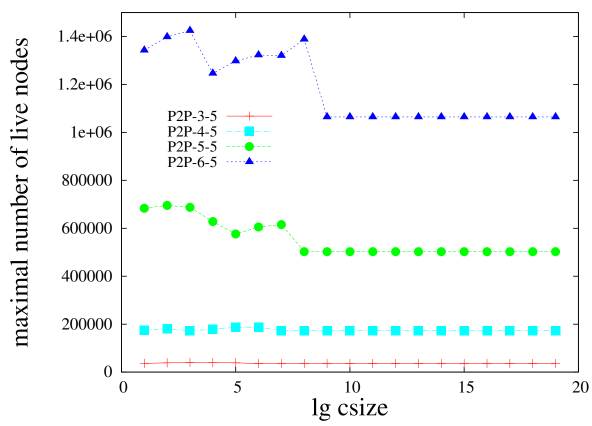
[17] Y.-T. Lai, S. Sastry, Edge-valued binary decision diagrams for multi-level hierarchical verification, in: Proc. of the 1992 Design Automation Conference (DAC), IEEE CS Press, 1992, pp. 608–613.



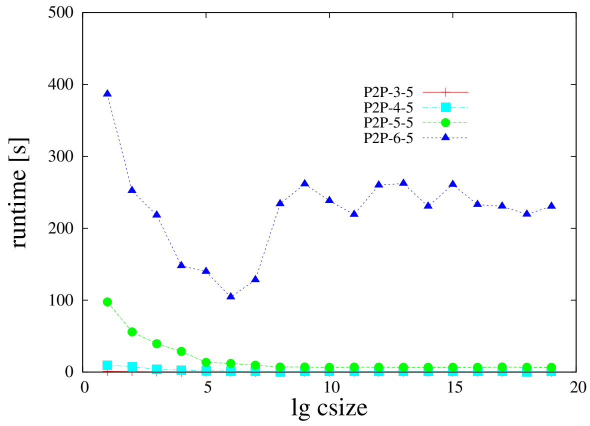
(a) FPCS (Peak # of nodes)



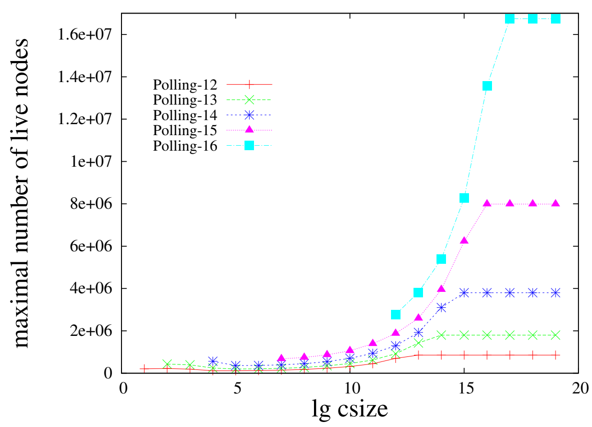
(b) FPCS (Running time)



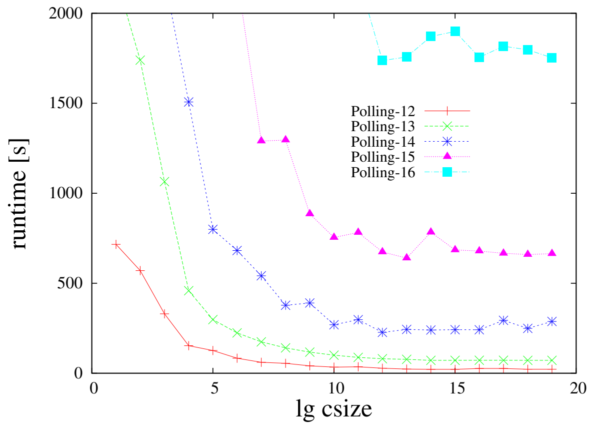
(c) P2P (Peak # of nodes)



(d) P2P (Running time)

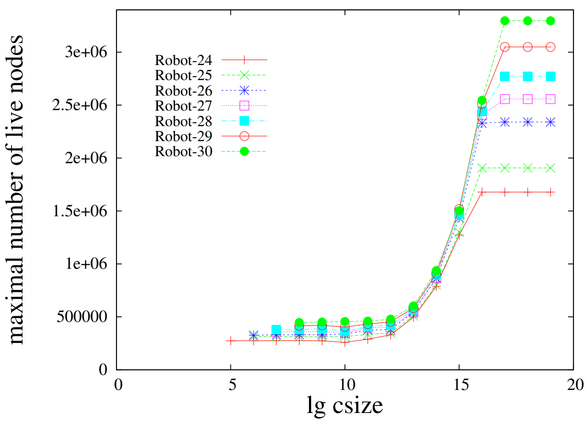


(e) Polling system (Peak # of nodes)

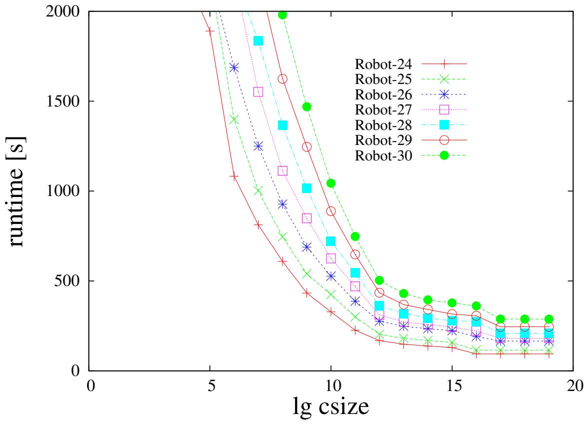


(f) Polling system (Running time)

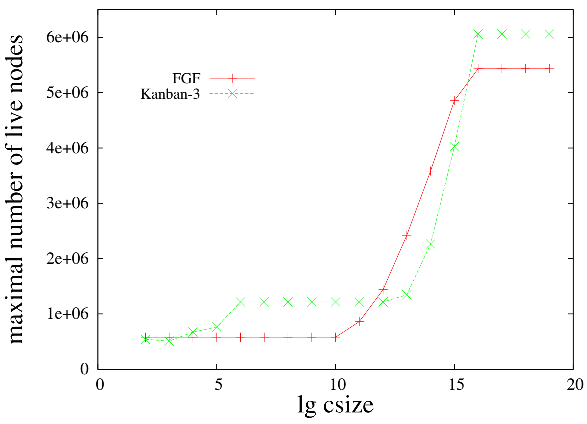
Figure 3: Maximal number of live nodes and running times of the hybrid algorithm for different chunk sizes (part 1)



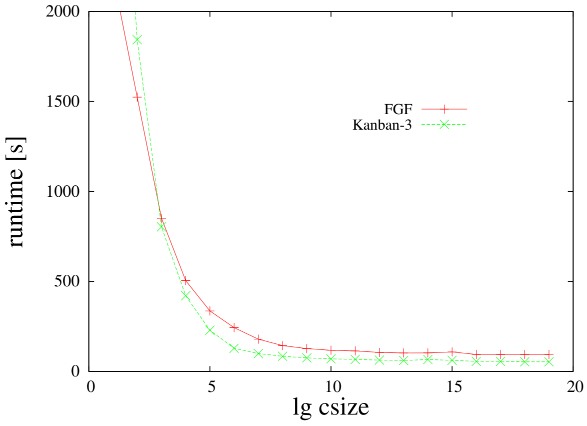
(a) Grid World Robot (Peak # of nodes)



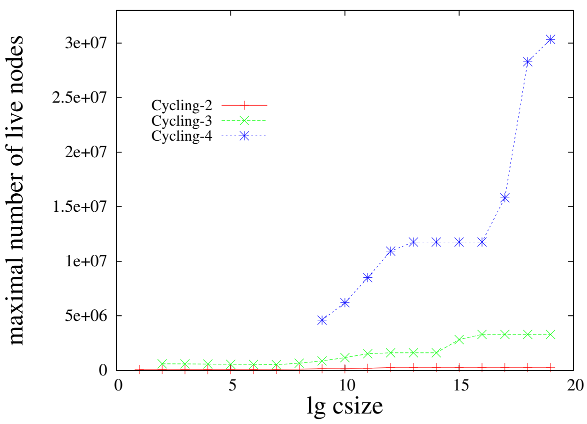
(b) Grid World Robot (Running time)



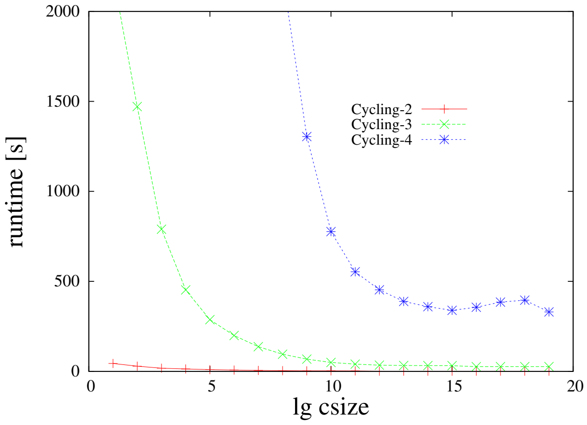
(c) FGF Signalling / Kanban Production System (Peak # of nodes)



(d) FGF Signalling / Kanban Production System (Running time)



(e) Cell Cycle Control (Peak # of nodes)



(f) Cell Cycle Control (Running time)

Figure 4: Maximal number of live nodes and running times of the hybrid algorithm for different chunk sizes (part 2)

[18] Y.-T. Lai, M. Pedram, S. B. K. Vrudhula, Formal verification using edge-valued binary decision diagrams, IEEE Trans. Computers 45 (2) (1996) 247–255.

[19] S. Minato, Zero-suppressed BDDs for set manipulation in combinatorial problems, in: Proc. of the 1993 Design Automation Conference (DAC), 1993, pp. 272–277.

[20] G. Ciardo, A. S. Miner, A data structure for the efficient Kronecker solution of GSPNs, in: 8$^{\text{th}}$ International Workshop on Petri Nets Performance Models (PNPM), 1999, pp. 22–31.

[21] A. Hinton, M. Kwiatkowska, G. Norman, D. Parker, PRISM: A tool for automatic verification of probabilistic systems, in: Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Anlysis of Systems (TACAS), Vol. 3920 of Lecture Notes in Computer Science, 2006, pp. 441–444.

[22] M. Kuntz, M. Siegle, E. Werner, Symbolic performance and dependability evaluation with the tool CASPA, in: FORTE Workshops 2004, Vol. 3236 of Lecture Notes in Computer Science, Springer, 2004, pp. 293–307.

[23] E. Böde, M. Herbstritt, H. Hermanns, S. Johr, T. Peikenkamp, R. Pulungan, J. Rakow, R. Wimmer, B. Becker, Compositional dependability evaluation for STATEMATE, IEEE Transactions on Software Engineering 35 (3) (2009) 274–292.

[24] R. Enders, T. Filkorn, D. Taubner, Generating BDDs for symbolic model checking in CCS, Distributed Computing 6 (3) (1993) 155–164.

[25] H. Hermanns, J. Meyer-Kayser, M. Siegle, Multi terminal binary decision diagrams to represent and analyse continuous time markov chains, in: B. Plateau, W. J. Stewart, M. Silva (Eds.), 3rd Int'l Workshop on the Numerical Solution of Markov Chains (NSMC), Prensas Universitarias de Zaragoza, Zaragoza, Spain, 1999, pp. 188–207.

[26] H. Hermanns, M. Z. Kwiatkowska, G. Norman, D. Parker, M. Siegle, On the use of MTBDDs for performability analysis and verification of stochastic systems, Journal of Logic and Algebraic Programming 56 (1–2) (2003) 23–67.

[27] M. Kwiatkowska, G. Norman, D. Parker, Probabilistic symbolic model checking with PRISM: A hybrid approach, Software Tools for Technology Transfer 6 (2) (2004) 128–142.

[28] S. Blom, S. Orzan, Distributed branching bisimulation reduction of state spaces, in: Proc. of the Int'l Workshop on Parallel and Distributed Model Checking, Vol. 89 of Electronic Notes in Theoretical Computer Science, Elsevier, 2003, pp. 99–113.

[29] S. Blom, S. Orzan, A distributed algorithm for strong bisimulation reduction of state spaces, Software Tools for Technology Transfer 7 (1) (2005) 74–86.

[30] S. Blom, S. Orzan, Distributed state space minimization, Software Tools for Technology Transfer 7 (3) (2005) 280–291.

[31] R. Wimmer, M. Herbstritt, H. Hermanns, K. Strampp, B. Becker, Sigref – A symbolic bisimulation tool box, in: Proc. of the Int'l Symp. on Automated Technology for Verification and Analysis (ATVA), Vol. 4218 of Lecture Notes in Computer Science, 2006, pp. 477–492.

[32] S. Derisavi, A symbolic algorithm for optimal Markov chain lumping, in: Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Anlysis of Systems (TACAS), Lecture Notes in Computer Science, 2007, pp. 139–154.

[33] S. Derisavi, Signature-based symbolic algorithm for optimal Markov chain lumping, in: Proc. of the Int'l Conf. on Quantitative Evaluation of Systems (QEST), 2007, pp. 141–150.

[34] E. Böde, M. Herbstritt, H. Hermanns, S. Johr, T. Peikenkamp, R. Pulungan, R. Wimmer, B. Becker, Compositional performability evaluation for statemate, in: Proc. of the Int'l Conf. on Quantitative Evaluation of Systems (QEST), 2006, pp. 167–178.

[35] P. J. Schweitzer, Aggregation methods for large Markov chains, in: G. Iazeolla, P.-J. Courtois, A. Hordijk (Eds.), Proc. of the Int'l Workshop on Computer Performance and Reliability, Elsevier, 1984, pp. 275–286.

[36] S. Baarir, C. Dutheillet, S. Haddad, J.-M. Ilié, On the use of exact lumpability in partially symmetrical well-formed nets, in: Proc. of the Int'l Conf. on Quantitative Evaluation of Systems (QEST), IEEE CS Press, 2005, pp. 23–32.

[37] M. Bozga, O. Maler, On the representation of probabilities over structured domains, in: N. Halbwachs, D. Peled (Eds.), Proc. of the Int'l Conf. on Computer-Aided Verification (CAV), Vol. 1633 of Lecture Notes in Computer Science, Springer, Trento, Italy, 1999, pp. 261–273.

[38] K. Lampka, M. Siegle, Analysis of Markov reward models using zero-suppressed multi-terminal BDDs, in: L. Lenzini, R. L. Cruz (Eds.), Proc. of the 1st Int'l Conf. Performance Evaluation Methodolgies and Tools (VALUETOOLS), Vol. 180 of ACM International Conference Proceeding Series, ACM, 2006, p. 35.

[39] I. Wegener, Branching programs and binary decision diagrams, SIAM Monographs on Discrete Mathematics and Applications, SIAM, 2000.

[40] H. Hermanns, M. Z. Kwiatkowska, G. Norman, D. Parker, M. Siegle, On the use of MTBDDs for performability analysis and verification of stochastic systems, Journal of Logic and Algebraic Programming 56 (1-2) (2003) 23–67.

[41] J. E. Hopcroft, An $n \log n$ algorithm for minimizing states in a finite automaton, in: Z. Kohavi, A. Paz (Eds.), Theory of Machines and Computations, Academic Press, 1971, pp. 189–196.

[42] S. Derisavi, Solution of large Markov models using lumping techniques and symbolic data structures, Ph.D. thesis, University of Illinois at Urbana-Champaign (2005).

[43] A. Bouali, R. de Simone, Symbolic bisimulation minimisation, in: Proc. of the Int'l Conf. on Computer-Aided Verification (CAV), Vol. 663 of Lecture Notes in Computer Science, 1992, pp. 96–108.

[44] R. Wimmer, M. Herbstritt, B. Becker, Optimization techniques for BDD-based bisimulation minimization, in: ACM Great Lakes Symposium on VLSI, 2007, pp. 405–410.

[45] F. Somenzi, CUDD: Colorado University decision diagram package. public software, Colorado University, Boulder, `http://vlsi.colorado.edu/~fabio/` (2007).

[46] W. H. Sanders, L. M. Malhis, Dependability evaluation using composed SAN-based reward models., Journal Parallel and Distributed Computing 15 (3) (1992) 238–254.

[47] O. Ibe, K. Trivedi, Stochastic Petri net models of polling systems, IEEE Journal on Selected Areas in Communications 8 (9) (1990) 1649–1657.

[48] H. Younes, M. Kwiatkowska, G. Norman, D. Parker, Numerical vs. statistical probabilistic model checking, Software Tools for Technology Transfer 8 (3) (2006) 216–228.

[49] G. Ciardo, M. Tilgner, On the use of Kronecker operators for the solution of generalized stocastic Petri nets, ICASE Report 96-35, Institute for Computer Applications in Science and Engineering (1996).

[50] J. Heath, M. Kwiatkowska, G. Norman, D. Parker, O. Tymchyshyn, Probabilistic model checking of complex biological pathways, Theoretical Computer Science 319 (3) (2008) 239–257.

[51] P. Lecca, C. Priami, Cell cycle control in eukaryotes: A BioSpi model, Electronic Notes in Theoretical Computer Science 180 (3) (2007) 51–63.

[52] H. Hermanns, M. Siegle, Bisimulation algorithms for stochastic process algebras and their BDD-based implementation, in: Proc. of the 5th Int'l AMAST Workshop on Real-Time and Probabilistic Systems (ARTS'99), Vol. 1601 of Lecture Notes in Computer Science, 1999, pp. 144–264.

Ralf Wimmer received his diploma in computer science from the Albert-Ludwigs-University, Freiburg (Germany) in 2004. Since 2005 he has been working as a PhD student at the German Transregional Collaborative Research Center AVACS in Freiburg. His research interests include applications of symbolic methods for stochastic verification.

Salem Derisavi received his bachelor degree in computer engineering in 1999 from Sharif University of Technology, Iran and his PhD in computer science from the University of Illinois at Urbana-Champaign in 2005. He is currently working with the IBM Software Laboratory in Toronto, Canada. His research interests include designing efficient data structures and algorithms for functional and numerical analysis of finite-state models, and more recently, parallelizing compilers for multi-core processors.

Holger Hermanns studied at the University of Bordeaux, France, and the University of Erlangen/Nürnberg, Germany, where he received a diploma degree in computer science in 1993 (with honors) and a PhD degree from the Department of Computer Science in 1998 (with honors). From 1998 to 2006 he has been with the University of Twente, The Netherlands, holding an associate professor position since October 2001. Since 2003 he heads the Dependable Systems and Software Group at Saarland University, Germany. He has published more than 100 scientific papers, holds various research grants, and has co-chaired several international conferences including CAV, CONCUR and TACAS. His research interests include modeling and verification of concurrent systems, resource-aware embedded systems, and compositional performance and dependability evaluation.