PDMC 2008

Distributed Markovian Bisimulation Reduction aimed at CSL Model Checking[⋆]

Stefan Blom[1] Boudewijn R. Haverkort[2] Matthias Kuntz[3]
Jaco van de Pol[4]

University of Twente
Faculty for Electrical Engineering, Mathematics and Computer Science
Enschede, The Netherlands

Abstract

The verification of quantitative aspects like performance and dependability by means of model checking has become an important and vivid area of research over the past decade.
An important result of that research is the logic CSL (continuous stochastic logic) and its corresponding model checking algorithms. The evaluation of properties expressed in CSL makes it necessary to solve large systems of linear (differential) equations, usually by means of numerical analysis. Both the inherent time and space complexity of the numerical algorithms make it practically infeasible to model check systems with more than 100 million states, whereas realistic system models may have billions of states.
To overcome this severe restriction, it is important to be able to replace the original state space with a probabilistically equivalent, but smaller one. The most prominent equivalence relation is bisimulation, for which also a stochastic variant exists (Markovian bisimulation). In many cases, this bisimulation allows for a substantial reduction of the state space size. But, these savings in space come at the cost of an increased time complexity. Therefore in this paper a new distributed signature-based algorithm for the computation of the bisimulation quotient of a given state space is introduced.
To demonstrate the feasibility of our approach in both a sequential, and more important, in a distributed setting, we have performed a number of case studies.

Keywords: Distributed bisimulation reduction, Markovian bisimulation, Stochastic model checking

1 Introduction

It is extremely important to develop techniques that allow the construction and analysis of distributed computer and communication systems. These systems must work correctly and meet high performance and dependability requirements. Using stochastic model checking, it is possible to perform a combined analysis of both qualitative (correctness) and quantitative (performance and dependability) aspects

[⋆] This work has been partially funded by the Netherlands Organisation for Scientific Research (NWO) under FOCUS/BRICKS grant number 642.000.504 (VeriGem) and by the EU under grant number FP6-NEST STREP 043235 (EC-MOAN).

[1] Email: sccblom@cs.utwente.nl
[2] Email: brh@cs.utwente.nl
[3] Email: kuntzwgm@cs.utwente.nl
[4] Email: vdpol@cs.utwente.nl

of a system model. Models that incorporate both qualitative and quantitative aspects of system behaviour can be described using various high-level formalisms, such as stochastic process algebras [17,14], stochastic Petri nets [1] and stochastic activity networks [25] (SANs).

In order to model check stochastic systems, over the last years a number of stochastic extensions of the logic CTL [7] have been devised. The most prominent extension is the logic CSL [3] (continuous stochastic logic). The applicability of stochastic model checking is limited by the complexity, i.e., the size of system models that are to be verified. At the heart of stochastic model checking lies the solution of huge sparse systems of linear (differential) equations. This limits the size of systems that can be practically analysed to some 10^8 states. To overcome these limitations we can think of several approaches. An important approach in this context is to reduce the state space size by the use of a notion of Markovian bisimulation.

We are aware of several approaches to reduce a given transition system with respect to a Markovian bisimulation. In the stochastic process algebra tool TIPP [15] an algorithm for Markovian bisimulation reduction based on the classical partition refinement algorithms [24,20,10] has been used. The `bcg_min` tool in the CADP toolset also supports the minimisation of transition systems with respect to Markovian bisimulation [12]. In [16] symbolic implementations, i.e., implementations that rely on multi-terminal binary decision diagrams [11] (MTBBDs) are used. More recently, in [8], a symbolic approach for signature-based [5] computation of the Markovian bisimulation quotient is presented. Only in [8] it is possible to use state labels which is necessary for model checking CSL formulae. However, no Markovian bisimulation for CSL model checking was applied in [8]. For CSL model checking a variant of Markovian bisimulation, Markov-AP bisimulation [3], has been introduced. For Markov-AP bisimulation, the atomic propositions (APs) that hold in a state are additionally taken into account. The only approach that actually applies a notion of bisimulation suited for CSL model checking is reported in [21]. The authors use the bisimulation algorithm of [9]. The drawback of all these approaches is their high time and memory complexity.

Therefore, in this work we propose a distributed signature-based Markov-AP bisimulation algorithm. In contrast to [21] we apply a signature-based algorithm [5] which is more memory-efficient than the algorithm from [9] used in [21]. In some cases, our algorithm is also faster than the algorithm applied in [21]. Furthermore, we are not aware of any approach that computes the quotient of any notion of a stochastic bisimulation relation in a distributed way.

The paper is further organised as follows. In Section 2 we introduce the syntax and semantics of CSL, the notion of Markov-AP bisimulation, as well as a signature-based definition of Markov-AP bisimulation. In Section 3 a distributed implementation for the signature-based computation of the Markov-AP bisimulation quotient is presented. In Section 4 we present a few widely used case studies in the realm of stochastic verification. These case studies are used in Sec. 5 and Sec. 6 to evaluate the efficiency of the sequential and the distributed signature-based tools, respectively. The paper ends with a summary and outlook.

2 CSL and Markovian Bisimulation

In this section we introduce the syntax and semantics of CSL, as well as a bisimulation relation that preserves the validity of CSL formulae.

2.1 Syntax and Semantics of CSL

The logic CSL [2,3] provides means to express and verify performance and dependability properties.

2.1.1 Syntax of CSL

The logic CSL extends CTL by replacing the untimed next (X) and until (U) operator with timed variants; it replaces the path quantifiers E and A with a probabilistic variant $\mathcal{P}_{\bowtie p}$, to reason about the probabilities with which a path formula is satisfied in the given model. Finally, CSL provides a steady-state operator $\mathcal{S}_{\bowtie p}$ to reason about the the stationary system behaviour.

Formally, the syntax of CSL can be defined as follows:

Definition 2.1 [Syntax of CSL] Let $p \in [0,1]$, $q \in AP$, and $\bowtie \in \{\leq, <, \geq, >\}$. State formulae of CSL are defined by the following grammar:

$$\Phi := q \;\big|\; \neg\Phi \;\big|\; \Phi \vee \Phi \;\big|\; \mathcal{S}_{\bowtie p}(\Phi) \;\big|\; \mathcal{P}_{\bowtie p}(\phi)$$

where ϕ is a path formula that is defined as follows:

$$\mathsf{X}^I \Phi \;\big|\; \Phi \,\mathsf{U}^I\, \Phi$$

$I = [t, t']$ is a closed time interval with $t \geq 0$ and $t' \neq 0$.

2.1.2 Semantics of CSL

At first, we have to introduce the semantic model of CSL, which is a state-labelled continuous-time Markov chain (SMC).

Definition 2.2 [State-Labelled Continuous-Time Markov Chains] A state labelled continuous-time Markov chain (SMC) $\mathcal{M}$ is a triple:

$$\mathcal{M} = (S, R, L)$$

where:

- S is a finite set of states.
- $R \subseteq S \times I\!R \times S$ is the transition relation.

  If $(s, \lambda, s') \in R$, we write $s \xrightarrow{\lambda} s'$ and $\lambda \in I\!R$ is the rate with which a state transition occurs, i.e. λ is the transition rate from state s to s'.
- $L : S \mapsto 2^{AP}$ a state labelling function that associates with each state a set of atomic propositions that are true in this state.

Definition 2.3 [Paths in SMCs] Let $\mathcal{M}$ be an SMC. An infinite path σ is a sequence $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} ...s_i \xrightarrow{t_i}$, where for all $i \in I\!N$, $s_i \in S$, $t_i \in I\!R$, is the actual sojourn time in state s_i, A finite path is a sequence $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} ...s_{n-1} \xrightarrow{t_{n-1}} s_n$ such that s_n is absorbing.

$\mathsf{PATH}^{\mathcal{M}}$ is the set of all finite and infinite paths in $\mathcal{M}$, $\mathsf{PATH}^{\mathcal{M}}(s)$ is the set of all paths with initial state s.

Due to lack of space, we will provide only an informal semantics of CSL. For a more detailed account of the semantics of CSL, we refer to [3]. Intuitively, the CSL formulae have the following meaning:

- $\mathcal{S}_{\bowtie p}(\Phi)$: The stationary state probability to be in a state that satisfies Φ is within $\bowtie p$.
- $\mathcal{P}_{\bowtie p}(\phi)$: The probability measure of all paths, satisfying ϕ conforms to the bounds imposed by $\bowtie p$.
- $\mathsf{X}^I\phi$ is the time bounded variant of the next-operator known from CTL. The path formula $\mathsf{X}^I\phi$ expresses that a state that satisfies ϕ must be reached after a passage of at least t and at most t' time units, if $I = [t, t']$.
- $\Phi\ \mathsf{U}^I\ \Psi$ is the time bounded variant of the CTL until-operator. $\Phi\ \mathsf{U}^I\ \Psi$ expresses that a state satisfying Ψ must be reached at a time point that lies within $I = [t, t']$. All states visited before such a Ψ state must satisfy Φ.

2.2 Bisimulation for CSL

To combat the notorious state space explosion problem, equivalence relations like bisimulation in the functional world or Markovian bisimulation in the Markovian world have turned out to be quite useful. In the context of SMC a variant of Markovian bisimulation, Markov-AP bisimulation, can be defined, to reduce the state space of SMCs thereby preserving the validity of CSL formulae that are valid on the unreduced SMC.

Definition 2.4 [Markov-AP bisimulation] Given an SMC $\mathcal{M} \equiv (S, R, L)$, an equivalence relation $\mathcal{B} \subseteq S \times S$ is a Markov-AP bisimulation over $\mathcal{M}$ if $(s, s') \in \mathcal{B}$ implies

(i) $L(s) = L(s')$, and

(ii) $\mathbf{R}(s, C) = \mathbf{R}(s', C)$ for all $C \in S/\mathcal{B}$

with $\mathbf{R}(s, C) = \sum_{s' \in C} \mathbf{R}(s, s')$, $S/\mathcal{B} = \{C_1, C_2, ..., C_n\}$ is the partitioning of S into its equivalence classes C_i with respect to $\mathcal{B}$, $\mathbf{R}(s, s')$ denotes the transition rate from state s to state s'. Two states s and s' are Markov-AP bisimilar, if there is a Markov-AP bisimulation $\mathcal{B}$ such that $(s, s') \in \mathcal{B}$.

Definition 2.5 Let $[s]_{\mathcal{B}}$ be the equivalence class of s with respect to Markov-AP bisimulation relation $\mathcal{B}$, then the SMC $\mathcal{M}/\mathcal{B}$ is defined as follows:

$$\mathcal{M}/\mathcal{B} = (S/\mathcal{B}, R_{\mathcal{B}}, L_{\mathcal{B}})$$

where

- $S/\mathcal{B} = \{C_1, C_2, ..., C_n\}$
- $\mathbf{R}_{\mathcal{B}}([s]_{\mathcal{B}}, C) = \mathbf{R}(s, C)$
- $L_{\mathcal{B}}([s]_{\mathcal{B}}) = L(s)$

Without proof, we cite the following theorem [3], which states that $\mathcal{M}/\mathcal{B}$ satisfies the same CSL properties as $\mathcal{M}$:

Theorem 2.6 *Markov-AP bisimulation preserves the validity of CSL formulae; i.e. let $\mathcal{B}$ be a Markov-AP bisimulation relation, then it holds for every CSL formula:*

$$\mathcal{M}, s \models \Phi \Leftrightarrow \mathcal{M}/\mathcal{B}, [s]_{\mathcal{B}} \models \Phi$$

For the computation of the bisimulation quotient it is crucial to choose an initial partition. For Markov-AP bisimulation we choose to initially group those states together that bear the same state labelling. Starting from this initial partition, the partitioning is refined during the computation, until no further refinement can be found, i.e. a fixed-point is reached.

2.3 Signature-based Markov-AP Bisimulation

The signature of a state with respect to an equivalence relation is supposed to reflect the transitions the state has into the various equivalence classes. For Markov(-AP) bisimulation these transitions are the rates of the states into the classes:

Definition 2.7 Given an SMC $\mathcal{M}$ and an equivalence relation $\mathcal{B}$ on S, the Markov signature of a state $s \in S$ with respect to $\mathcal{B}$ is given by

$$\mathsf{sig}(s, \mathcal{B}) = \{(\mathbf{R}(s, [s']_{\mathcal{B}}), [s']_{\mathcal{B}}) \mid s \rightarrow s'\}$$

Definition 2.8 Given an equivalence relation $\mathcal{B}$ on S, we define the refinement of $\mathcal{B}$ as the equivalence relation $\mathsf{sig}[\mathcal{B}]$, such that

$$(s, s') \in \mathsf{sig}[\mathcal{B}] \Leftrightarrow (s, s') \in \mathcal{B} \wedge \mathsf{sig}(s, \mathcal{B}) = \mathsf{sig}(s', \mathcal{B})$$

We say that $\mathcal{B}$ is *stable* if $\mathsf{sig}[\mathcal{B}] = \mathcal{B}$. The coarsest stable equivalence refining $\mathcal{I}$ is computed by the following basic algorithm for signature refinement:

```
$\mathsf{sig}^*[\mathcal{I}]$ :=
    $\mathcal{B}$ := $\mathcal{I}$
    loop
        $\mathcal{B}'$ := $\mathsf{sig}[\mathcal{B}]$
        if $\mathcal{B}'$ == $\mathcal{B}$ return $\mathcal{B}$
        $\mathcal{B}$ := $\mathcal{B}'$
```

The Markov signature yields Markov-AP bisimulation reduction if the initial partition is chosen to be label-equivalence:

Theorem 2.9 *For finite SMC's the coarsest possible Markov-AP bisimulation is equal to* $\mathsf{sig}^*[\mathcal{B}]$*, where* $(s, s') \in \mathcal{B} \Leftrightarrow L(s) = L(s')$.

Proof. The proof proceeds along the same lines as in [4]. □

3 Distributed Implementation of Signature-based Markov-AP Bisimulation

In this section, we describe how to compute $\mathsf{sig}^*[\mathcal{B}]$ in a distributed setting. The algorithm used is a variant of the algorithm described in [4]. First, we assume

Markov Chain:

label$[N_i]$	array storing the atomic propositions holding in a state. Sets of atomic propositions are represented as 32 bit bitsets.
inedge$[_{j=0}^{W-1}][N_{j\,i}]$	array of arrays where inedge$[j][e]$ stores the destination state of edge number e from j to i.
rate$[\Sigma_{j=0}^{W-1} N_{i\,j}]$	array storing the rates of outgoing edges.
destworker$[\Sigma_{j=0}^{W-1} N_{i\,j}]$	array storing the destination worker of outgoing edges.
destedge$[\Sigma_{j=0}^{W-1} N_{i\,j}]$	array storing the destination edge number of outgoing edges.
begin$[N_i + 1]$	array where begin$[s]$ is the index in rate, destworker and destedge of the first edge of state s.

Equivalence Classes:

stateclass$[N_i]$	array storing the equivalence class of local states
edgeclass$[_{j=0}^{W-1}][N_{i\,j}]$	array of arrays where edgeclass$[j][e]$ stores the class of the destination state of edge e form i to j.

Locked Data Structures:

queue[W]	internal queue to match submitted hash table request with replies
hashtable	hash table that maps pairs of class numbers and signatures to equivalence class numbers

Table 1
Distributed data structures

that we can compute a globally unique byte string representation of signatures and explain distributed refinement. Second, we explain how to compute such a globally unique byte string.

We start by explaining how the SMC is distributed. There are W workers which are numbered $0, \ldots, W-1$. The states are distributed evenly over the workers, where N_i is the number of states of worker i. These states are numbered from 0 to $N_i - 1$. The distribution of states over sets $(S_i)_{i=0}^{W-1}$ leads to a distribution of edges over the sets $(E_{i\,j})_{i=0}^{W-1}{}_{j=0}^{W-1}$. The edges between each pair of workers i and j are numbered from 0 to $|E_{i\,j}|-1$. At each worker, we store the atomic propositions of its states, the rates of the outgoing transitions, the edge numbers of the outgoing edges and the destination states of the incoming edges. Thus two workers can exchange information along an edge by referring to the number of the edge. See the first part of Table 1. This table contains a listing of variables. In case of array variables the dimensions are included. Note that, we have two arrays of arrays in which the lengths of the inner arrays depend on the index of the first dimension. For example, a 4×4 triangle x would be written down as x$[_{i=1}^{4}][i]$.

We continue by explaining the reduction algorithm in Table 2. Equivalence relations on states are stored by numbering equivalence classes and storing that number in an array, cf. the middle of Table 1. To compute the refinement $\mathsf{sig}[\,\mathcal{B}\,]$ of $\mathcal{B}$, we must compute the signatures and find new equivalence class numbers.

```
var $me$ // identity of worker
reduce()
    for $(i = 0; i < N_{me}; i ++)$ stateclass[$i$]=label[$i$];
    repeat
        // First exchange class numbers:
        send-block() $\parallel \left( \overset{W-1}{\underset{w=0}{\parallel}} \text{receive-block}(w) \right)$
        // Then compute new class numbers:
        reset hashtable
        compute-signatures() $\parallel \left( \overset{W-1}{\underset{w=0}{\parallel}} \text{hash-server}(w) \right) \parallel \left( \overset{W-1}{\underset{w=0}{\parallel}} \text{receive-index}(w) \right)$
    until stable
send-block()
    for$(i = 0; i < \max(N_{*\,me}); i ++)$
        for$(w = 0; w < W; w ++)$ if$(i < N_{w\,me})$
            send stateclass[inedge[$w$][$i$]] to $w$
receive-block($w$)
    for$(i = 0; N_{me\,w}; i ++)$ receive edgeclass[$w$][$i$] from $w$
compute-signatures()
    for$(i = 0; i < N_{me}; i ++)$
        $sig$=pair(stateclass[$i$],signature($i$));
        send $sig$ to who($sig$)
        put i in queue[who($sig$)]
    for$(w = 0; w < W; w ++)$
        send empty to $w$
        put -1 in queue[$w$]
hash-server($w$)
    loop
        receive $sig$ from $w$
        if ($sig$ == empty) return
        let pos = position of $sig$ in hashtable
        send pos to $w$
receive-index($w$)
    loop
        get i from queue[$w$]
        if $(i < 0)$ return
        receive stateclass[$i$] from $w$
```

Table 2
Distributed signature refinement algorithm

To compute the signature of a state, a worker needs to know the equivalence class numbers of all successor states. Because these successor states might be on a different worker, we first need to communicate this information. That is, the equivalence class number of every state must be sent to (the worker owning) any predecessor of that state. Computing new equivalence class numbers is achieved by inserting pairs of old equivalence class numbers and signatures (strings of bytes) into a global distributed hash table. The test $\mathcal{B} == \mathcal{B}'$ is implemented by counting the number of classes in B' and $\mathcal{B}$. Because we know that $\mathcal{B}'$ is a refinement of $\mathcal{B}$, we know that $\mathcal{B} == \mathcal{B}'$ is true if $|\mathcal{B}'| = |\mathcal{B}|$.

To avoid latency problems, the distributed hash table works asynchronously. Given a string of bytes and an address for the result, it is decided which worker is responsible for that string. Then the string is sent to that worker, the address is written into a FIFO queue and we return immediately. For each of the FIFO queues there is a thread that reads addresses, waits for the response from the remote worker and writes the responses to the correct addresses.

As described above the algorithm uses many small messages. In practice these messages are treated as a continuous stream of data which is sent in blocks of multiple kilobytes.

Representing a signature for Markov(-AP) bisimulation as a globally unique byte string is not a trivial exercise: to compute signatures we need to add a small number of rates. If we represent the rates as floating point numbers then we have to deal with errors in the sums, which means that we cannot use the sums directly in the byte string representation because signature equality is decided by comparing the byte strings.

We have implemented two solutions to this problem. The first solution uses floats for the rates and rounds the resulting sums to get a unique representation. The second solution translates the given rates to rational numbers and thus eliminates the errors from the sums. Neither solution is perfect: using rounding it is possible to create an example where the true value of a sum is a boundary value for rounding and where some sums add to just below the boundary and others to just above, resulting in two distinct signatures for states which should have the same signature. Translating floats to rational numbers is tricky because we need to translate them in such a way that if we have two sequences, which as floats add up to the same value up to ϵ then they should add up to the same number as rational numbers. We used the latter solution in our implementation, but a detailed description goes beyond the scope of this paper.

4 Case Study Descriptions

We will now describe four case studies from the literature, with certain CSL properties that we want to be preserved by Markov-AP bisimulation.

4.1 Simple Peer-to-Peer Protocol

This case study is based on [23]. Here, a simple peer-to-peer protocol based on BitTorrent is described. $N+1$ clients try to download a file, that is divided into K blocks. In the initial state, there is one client that is in possession of the entire file, i.e., all K blocks, and N clients have no block at all. Each client can download a block from each other client. Here, we investigate a system with $K = 5$ blocks and $N = 2, 3, 4$ additional clients. A typical CSL property for the Peer-to-Peer Protocol is:

- Is the probability that all clients have received all blocks by time bound less than T larger than 99 percent?

4.2 Workstation Cluster

This case study is based on [13]. The system consists of two sub-clusters, where each sub-cluster possesses N workstations. The workstations in the respective sub-clusters are connected according to a star-topology with a central switch. The sub-cluster central switches communicate via a central backbone. All components are subject to failures and can be repaired. For all components a single repair unit is available. The employed repair strategy is *random*, i.e., if more than one component awaits repair, the repair unit chooses the component which is to be repaired next according to a typically uniform probability distribution.

For the Workstation Cluster, the CSL property of interest is:

- The system will always be able to offer premium QoS at some point in the future, where premium service means, that $\frac{3N}{4}$ workstations are operational and connected via switches and backbone.

4.3 Polling System

Here, a cyclic server-polling system with N stations is analysed. The model was introduced in [19]. The server polls the N stations in a cyclic way. After polling station i, station i is served. If station i is idle, it is skipped.

For the Polling System, we define a number of CSL requirements, the system has to satisfy:

(i) What is the probability that in the long run station 1 is awaiting service?

(ii) What is the probability that in the long run station 1 is idle?

(iii) Is the probability, once a station becomes full, it will eventually be polled above 90 percent?

(iv) Is the probability that from the inital state, station 1 is served before station 2 below 25 percent?

This leads to the formulae

(i) $\mathcal{S}_{\bowtie p}(s1 = 1\&!(s = 1\&a = 1))$

(ii) $\mathcal{S}_{\bowtie p}(s1 = 0)$

(iii) $\mathcal{P}_{\geq 0.9}(\mathsf{true}\ \mathsf{U}\ (s = 1\&a = 0))$

(iv) $\mathcal{P}_{<0.25}(!(s = 2\&a = 1)\ \mathsf{U}\ (s = 1\&a = 1))$

where $s1 = 1$, $(s = 1\&a = 1)$, etc. can be regarded as state labels from the high-level specification.

4.4 Kanban System

This case study was originally described in [6]. We model a Kanban system with four cells, a single type of Kanban cards, and the possibility that some workpiece

	N	unreduced SMC		reduced SMC		reduction time	
		states	transitions	states	transitions	mrmc	ltsmin
P2P	2	1,024	5,121	56	141	$< 0.1msec.$	$< 0.1msec.$
	3	32,768	245,761	792	3,961	0.12 sec.	0.19 sec.
	4	1,048,576	10,485,761	15,504	124,033	10.28 sec.	11.17 sec.
cluster	8	2,772	12,832	1,413	6,443	0.01 sec.	0.03
	16	10,132	48,160	5,117	24,131	0.04 sec.	0.42 sec.
	32	38,676	186,400	19,437	93,299	0.19 sec.	3.22 sec.
	64	151,060	733,216	75,725	366,803	0.96 sec.	24.78 sec.
	128	597,012	2,908,192	298,893	1,454,483	4.38 sec.	89.24 sec.
	256	2,373,652	11,583,520	1,187,597	5,792,531	20.79 sec.	793.78 sec.
polling	10	15,360	89,600	1,536	8,960	0.051 sec.	0.17 sec.
	12	73,728	503,808	6,144	41,984	0.624 sec.	1. 19 sec.
	14	344,064	2,695,168	24,576	192,512	5.51 sec.	7.53 sec.
	16	1,572,864	13,893,632	98,304	868,352	32.12 sec.	38.61 sec.
	18	7,077,888	69,599,232	393,216	3,866,624	218.66 sec.	277.39 sec.
	19	14,942,208	154,402,816	786,432	8,126,464	-	667.59 sec.
kanban	3	58,400	446,400	58,400	446,400	0.989 sec.	0.52 sec.
	4	454,475	3,979,850	454,475	3,979,850	11.9 sec.	5.75 sec.
	5	2,546,432	24,460,016	2,546,432	24,460,016	100.3 sec.	42.04 sec.

Table 3
Comparison of mrmc and ltsmin.

may need to be reworked. We use N to denote the number of cards in the system. For the Kanban system we have not specified CSL formulae.

5 Empirical Evaluation: Sequential Case

In this section we show the feasibility of our signature-based reduction algorithm by means of the case studies from Sec. 4. That is, we compare both the time and memory efficiency of a serial version of our algorithm, as implemented in ltsmin with that of mrmc [22,21]. The next section is devoted to the evaluation of the distributed version of our algorithm.

5.1 General Remarks

As a high-level tool for the specification of the models of Sec. 4 we used the tool PRISM[18]. Using the PRISM specification, we generated the SMC, and stored it in the so called tra-format [15]. This format is the input format for mrmc, so we decided to use it also for ltsmin, although the tra-format is text-based and hence IO-inefficient. PRISM was also used for the specification of CSL properties we defined over the models under analysis. The state labels, i.e. the atomic propositions, that guide the initial state space partitioning for Markov-AP bisimulation are also exported from PRISM and stored in a separate file, that is required by mrmc, thus again we decided to use the same for ltsmin.

In the sequel, we do not take into consideration the time for state space generation, or reading the SMC from disk. All run times mentioned are for the computation of the bisimulation quotient of the given SMC only.

All serial experiments were conducted on a dual Intel E5320 (quad core 1.86GHz) and 4 GB RAM, running SuSe Linux 10.2.

5.2 *Simple Peer-to-Peer Protocol*

As we can see from the first block of table 3, we obtain substantial savings in the state space if we compute the Markov-AP bisimulation quotient. If we compute the Markovian bisimulation quotient, these savings still increase, e.g. for $N = 4$, we can reduce the original over 1 million states large state space to only 126 states.

We can observe that in this case mrmc is slightly faster than ltsmin, but for $N = 4$ mrmc used a maximum of 289MB, whereas ltsmin required about 125 MB of main memory for the same system.

5.3 *Workstation Cluster*

From the second block of table 3 we can see, that we can save about one half of the state space, when applying Markov-AP bisimulation.

For this problem, we can see a clear run time advantage of mrmc over ltsmin. On the other hand, ltsmin requires much less memory than mrmc, e.g. for $N = 256$, mrmc has a peak memory requirement of 682 MB, whereas ltsmin only requires 132 MB.

The reason for the big difference in time is that for n states and m transitions the complexity of mrmc is $\mathcal{O}(m \log n)$, whereas the complexity of ltsmin is $\mathcal{O}(mI)$, where I is the number of iterations needed. Worst case I can be n, but in practice we have never encountered an example where I was worse than $\mathcal{O}(N)$. Because the cluster example does not grow as fast with N as the other examples this means that ltsmin is not as effective. For strong bisimulation it is known that in similar cases it is very effective to use incremental signature computation. It is future work to see if that carries over to Markov(-AP) bisimulation.

5.4 *Polling System*

If only a Markovian bisimulation quotient is computed, we obtain the state space reduction shown in the third block of table 3.

In the Polling System case, we can observe that mrmc has (slight) run-time advantages over ltsmin, but for peak memory usage, ltsmin is again less demanding then mrmc. For $N = 18$ mrmc used a maximum of 2GB, whereas ltsmin required about 779 MB of main memory for the same system. For $N = 19$ mrmc ran out of memory, and ltsmin required about 2 GB of main memory.

We have also computed the Markov-AP bisimulation quotient for all formulae from Section 4.3 in isolation. We observed that neither of the sets of state labels, that is induced by these formulae led to any reduction of the state space size.

In the case where we compute the Markov-AP bisimulation quotient, we observed that for $N = 18$ mrmc runs out of memory, whereas for ltsmin the peak memory consumption was about 960 MB, and for $N = 19$ about 2 GB (no table is included).

5.5 *Kanban System*

For the Kanban system we first computed the bisimulation quotient without state labellings, i.e. for a pure Markovian bisimulation. This led to no reduction of the state space size, therefore, it is obvious, that also no reduction can be expected,

	sequential		1 worker		2 workers		4 workers		8 workers	
	time	mem	time	mem	time	mem	time	mem	time	mem
polling 16	70	201	356	765	155	382	87	196	39	94
cluster 256	721	163	1646	613	1191	312	805	149	1126	72
peer-to-peer 4	10.7	132	13.4	330	7.8	164	4.0	89	2.6	44
Kanban 5	32	357	691	1491	198	714	70	349	27	167

Table 4
Wall clock *time* in seconds, maximum *mem*ory per worker in MB.

when reducing the state space with respect to some CSL formulae. In the last block of table 3 the run times for both mrmc and ltsmin can be found.

Not very surprising, when not taking different rates into account, i.e. if we compute a simple strong bisimulation quotient, the state space of the Kanban system is reducible. For example for $N = 3$ the state space can be reduced from 58,400 to 33,200 states.

In the Kanban case, it can be observed that ltsmin is superior to mrmc both in reduction times and memory usage. We observed that ltsmin uses less than one sixth of the maximum memory requirements of mrmc, e.g. ltsmin took 294 MB of main memory for $N = 5$, whereas mrmc had a peak memory requirement of 1.9 GB.

6 Empirical Evaluation: Distributed Case

To test our distributed implementation, we used 4 dual Xeon E5320 servers with 8GB each. As test cases we used the polling 16, cluster 256, peer-to-peer 4 and Kanban 5 problems. In each case the full set of atomic propositions was used during reduction. (So the results for cluster 256 cannot be compared to those in the last section.) Each of those problems was executed using 1, 2, 4 and 8 workers, where in the last case two workers had to run on one server. The results of those tests are enumerated and compared to ltsmin in Table 4. The run time info is visualised and compared to ltsmin and mrmc in Fig. 1.

It can be observed that for both time and memory, the distributed tool running with one worker is quite a bit more expensive than the sequential tool. A large part of these differences can be explained by the difference in data structures. First, the sequential tool stores the graph in such a way that it has access to all successor states, but not to predecessors. For n states and m edges it needs $n + 2m$ words of memory. Storing the old and new equivalence class numbers is done per state and requires $2n$ additional words. In a distributed setting, access to remote states requires remote memory access which we chose to avoid. That required storing m words of predecessor information in addition to the $n + 2m$ words for successor information. The requirements for equivalence classes increase from $2n$ to $n+m$ due to the fact that we need to store the equivalence class number for each transition rather than each state. (The sequential implementation can look in the array for the destination state, the distributed implementation cannot.) Thus we get a total of $3n + 2m$ for the sequential tool and $2n + 4m$ for the distributed one. Because m is often an order of magnitude larger, this explains a doubling of the size. The next big chunk of memory is because of storage of signatures. The sequential tool can represent a signature by a representing state and when needed can recompute

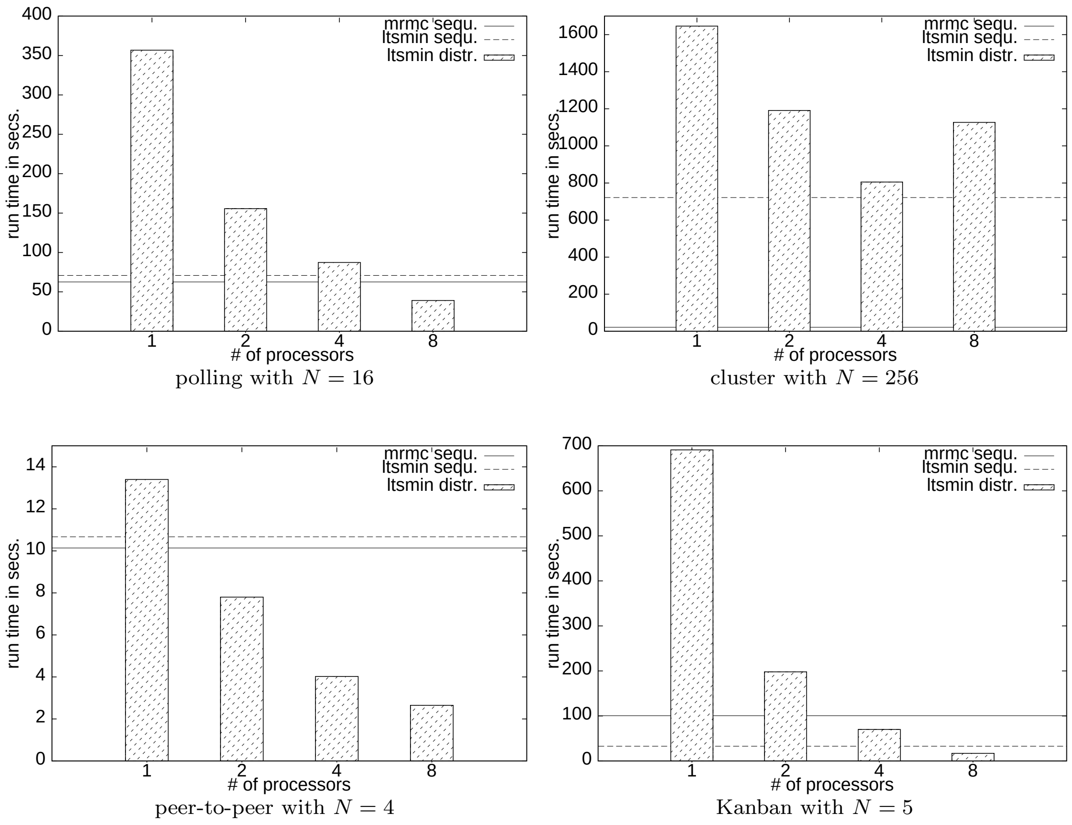

Fig. 1. Distributed run times (wall clock)

a signature from that state. The distributed tool cannot do that for its hash tables, because the owner of the hash table is typically not the owner of the representing state. If we have m' edges in the reduced system then this counts for $2m'$ additional words of memory. Finally, the distributed tool uses a cache to do remote lookups only once. This cache also requires memory.

Together, these differences explain why the single worker distributed tool uses 3 to 4 times as much memory as the sequential tool. Maintaining the extra data structures requires time, which explains part of the increase in time. Another part of the increase in time is due to the fact that it is much more expensive to send/receive data rather than write it directly. Note that the current structures were chosen to keep network usage low. At the price of increased network usage, the memory footprint can be reduced.

When comparing the run times for 1, 2, 4 and 8 workers it can be observed that memory usage decreases linearly and that wall time decreases super linearly with the exception of the cluster example.

In theory, the distributed tool scales up linearly in memory and time: the data structures do not require any duplication and although we have $\mathcal{O}(W^2)$ threads for W workers, all work must be initiated by a single main thread per worker. Moreover, nodes can only run a finite number of threads at the same time.

In practice, multiple threads can be both an advantage and a disadvantage. The advantage is that on modern multi-core machines one can exploit paralellism. The disadvantage is that each extra thread requires an overhead for synchronization and threads can contend for resources, such as caches. In addition, scheduling can have an effect if a thread gets blocked and we need to wait until it becomes active again. It is this scheduling effect that seems responsible for the increase in run time for the cluster experiment because we can see that there is a rather low CPU utilisation while the experiment runs. However, we need to perform more experiments before we can draw a firm conclusion.

7 Conclusion and Outlook

In this paper we have presented a signature-based variant of Markov-AP bisimulation and both a sequential and a distributed implementation.

In the sequential case, we have compared ltsmin with the tool mrmc because they compute exactly the same Markov-AP bisimulation quotient. In the near future, we will compare ltsmin with bcg_min as well. This is a somewhat difficult task due to the fact that bcg_min uses action labels rather than state labels. Thus, we need to be certain that any difference measured is due to a difference in implementation rather than a difference in encoding. In all studied cases, we could observe that mrmc has a two to five times higher peak memory requirement than ltsmin. In two cases (polling and peer-to-peer) the computation times of mrmc and ltsmin differed only slightly; in the case of the cluster system, mrmc was considerably faster than ltsmin and in the Kanban case study ltsmin was about two times faster than mrmc.

The distributed version of ltsmin showed that the signature-based algorithm scales nicely and in some cases even yielded superlinear speed-ups. Unfortunately, only for the peer-to-peer case study the times to compute the Markov-AP bisimulation quotient dropped below that of the sequential version when using two processors. In the remaining cases, eight processors were required to achieve a drop below the time of the sequential version. In one case we could even observe an increase in the computation time when switching from four to eight processors; this might be related to the long sequence of refinement steps.

In the near future we plan to optimise both the memory requirements and the computation times of our signature-based bisimulation reduction algorithm. For optimising memory requirements, we can think of several solutions. For example, we could perform the copying of the old class numbers on-the-fly rather than before the signature phase. This would reduce the storage requirements from one word per transition to one word per state. To improve the run times, we can gain by being more careful about how we use threads. Also, the current tool recomputes all signatures in every iteration. This is not needed: it suffices to compute the signatures that refer to states where the signatures changed in the previous iteration. We expect that this incremental computation technique will provide a huge increase in performance for the cluster example where we have a very high amount of iterations with few changes in nearly all of the steps.

References

[1] Ajmone Marsan, M., G. Balbo and G. Conte, *A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems*, ACM Transactions on Computer Systems **2** (1984), pp. 93–122.

[2] Aziz, A., K. Sanwal, V. Singhal and R. K. Brayton, *Verifying continuous time markov chains*, in: R. Alur and T. A. Henzinger, editors, *CAV*, Lecture Notes in Computer Science **1102** (1996), pp. 269–276.

[3] Baier, C., B. Haverkort, H. Hermanns and J. Katoen, *Model-Checking Algorithms for Continuous-Time Markov Chains*, IEEE Trans. Software Eng. **29** (2003), pp. 1–18.

[4] Blom, S. and S. Orzan, *A distributed algorithm for strong bisimulation reduction of state spaces.*, International Journal on Software Tools for Technology Transfer (STTT) **7** (2005), pp. 74–86.

[5] Blom, S. and S. Orzan, *Distributed state space minimization*, STTT **7** (2005), pp. 280–291.

[6] Ciardo, G. and M. Tilgner, *On the use of Kronecker operators for the solution of generalized stochastic Petri nets*, Technical Report 96-35, ICASE (1996).

[7] Clarke, E., E. Emerson and A. Sistla, *Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach*, in: *10th ACM Annual Symp. on Principles of Programming Languages*, 1983, pp. 117–126.

[8] Derisavi, S., *A Signature-based Algorithm for Optimal Markov Chain Lumping*, in: *Proceedings of the 4th International Conference on the Quantitative Evaluation of Systems (QEST 2007)*, Edinburgh, UK, 2007, pp. 259–260.

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

[10] Fernandez, J., *An Implementation of an Efficient Algorithm for Bisimulation Equivalence*, Science of Computer Programming **13** (1989), pp. 219–236.

[11] Fujita, M., P. McGeer and J.-Y. Yang, *Multi-terminal Binary Decision Diagrams: An efficient data structure for matrix representation*, Formal Methods in System Design **10** (1997), pp. 149–169.

[12] Garavel, H. and H. Hermanns, *On Combining Functional Verification and Performance Evaluation Using CADP*, in: *FME '02: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right* (2002), pp. 410–429.

[13] Haverkort, B. R., H. Hermanns and J.-P. Katoen, *On the Use of Model Checking Techniques for Dependability Evaluation*, in: *Proc. 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, Erlangen, Germany, 2000, pp. 228–237.

[14] Hermanns, H., U. Herzog and J.-P. Katoen, *Process algebra for performance evaluation*, Theoretical Computer Science **274** (2002), pp. 43–87.

[15] Hermanns, H., U. Herzog, U. Klehmet, V. Mertsiotakis and M. Siegle, *Compositional performance modelling with the TIPPtool*, Performance Evaluation **39** (2000), pp. 5–35.

[16] Hermanns, H. and M. Siegle, *Bisimulation Algorithms for Stochastic Process Algebras and their BDD-based Implementation*, in: J.-P. Katoen, editor, *ARTS'99, 5th Int. AMAST Workshop on Real-Time and Probabilistic Systems* (1999), pp. 144–264.

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

[18] Hinton, A., M. Kwiatkowska, G. Norman and D. Parker, *PRISM: A Tool for Automatic Verification of Probabilistic Systems*, in: H. Hermanns and J. Palsberg, editors, *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, LNCS **3920** (2006), pp. 441–444.

[19] Ibe, O. and K. Trivedi, *Stochastic Petri Net Models of Polling Systems*, IEEE Journal on Selected Areas in Communications **8** (1990), pp. 1649–1657.

[20] Kanellakis, P. and S. Smolka, *CCS Expressions, Finite State Processes, and Three Problems of Equivalence*, Information and Computation **86** (1990), pp. 43–68.

[21] Katoen, J.-P., T. Kemna, I. Zapreev and D. Jansen, *Bisimulation minimisation mostly speeds up probabilistic model checking*, in: *Proc. 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, LNCS 4424 (2007), pp. 76–92.

[22] Katoen, J.-P., M. Khattri and I. S. Zapreev, *A Markov reward model checker*, in: *Quantitative Evaluation of Systems (QEST)* (2005), pp. 243–244.

[23] Kwiatkowska, M., G. Norman and D. Parker, *Symmetry Reduction for Probabilistic Model Checking*, in: T. Ball and R. Jones, editors, *Proc. 18th International Conference on Computer Aided Verification (CAV'06)*, Lecture Notes in Computer Science **4114** (2006), pp. 234–248.

[24] Paige, R. and R. Tarjan, *Three Partition Refinement Algorithms*, SIAM Journal of Computing **16** (1987), pp. 973–989.

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