Int J Softw Tools Technol Transfer (2005) 7: 43–60 / Digital Object Identifier (DOI) 10.1007/s10009-003-0129-2

# Sequential and distributed model checking of Petri nets

**Alexander Bell, Boudewijn R. Haverkort**

Department of Computer Science, Laboratory for Performance Evaluation and Distributed Systems, RWTH Aachen, 52056 Aachen, Germany*
e-mail: brh@cs.utwente.nl, alexander.bell@math.utwente.nl

Published online: 6 April 2004 – 

**Abstract.** In this paper we present sequential as well as distributed algorithms for model checking computational tree logic over finite-state systems specified as Petri nets. The algorithms rely on an explicit representation of the system's state space but do not require the transition relation to be explicitly available; it is recomputed whenever required. This approach allows us to model check very large systems, with hundreds of millions of states, in a fast and efficient way. For the case studies addressed, the distributed algorithms scale very well, as they show efficiencies in the range of 60% to 95%, depending on the test cases and case studies at hand.

## 1 Introduction

Over the last decade, model checking has established itself as a very powerful technique for system validation. With model checking, a formally specified property is verified to hold (or not) in a system specified using a high-level system specification language by using exhaustive state space exploration techniques, combined with techniques to combat the state space explosion problem [14, 24, 26]. In this paper we focus on the use of computational tree logic (CTL) [15] to formally specify system properties; furthermore, we assume that the system of interest is described as a Petri net (PN). The latter choice is not fundamental to our approach, although it does impact the way we implement the algorithms. The adaptation of our algorithms to other specification languages can be performed easily, but this is beyond the scope of the current paper.

When model checking realistic systems, one usually encounters (at least) two problems: the size of the state space of the system being modeled is prohibitive, and the time required to check even simple properties is very large. Although good results have been obtained with symbolic (implicit) state space representations, e.g., using binary decision diagrams, the actual model checking algorithms often become slower when used in combination with such symbolic approaches. For that reason, we adhere to an explicit state space representation based on the use of hash tables; very good results have been obtained with that for pure state space generation purposes in the context of stochastic Petri nets [22]. Another reason to proceed with an explicit state space representation (instead of an implicit one) is the fact that we see CTL model checking of PNs as a first step toward CS(R)L model checking [2, 3] of *stochastic* Petri nets (see also Sect. 2). In such cases, we will need explicit approaches to enable the computation of state probabilities in the underlying stochastic model.

To tackle the state space explosion problem, the use of multiprocessor systems or workstation clusters also helps; these systems often exhibit a very large (distributed) main memory, and, furthermore, the large combined computational power of such systems helps in effectively reducing model checking time.

The aim of the current paper is to extend our previous work on distributed state space generation [22] by developing efficient (in terms of computation and communication) algorithms for model checking CTL expressions over PNs in a distributed fashion. Experimental results for models with hundred(s) of millions of states show that one can attain very good speedups and efficiencies when using sound parallel programming techniques.

The rest of this paper is organized as follows. In Sect. 2 we present preliminaries with respect to Petri nets, their representation, and the computation of successor and predecessor states. Then, in Sect. 3, we briefly introduce

* Boudewijn Haverkort is now with the University of Twente, Design and Analysis of Communication Systems, P.O. Box 217, 7500 AE Enschede, the Netherlands. Alexander Bell was supported by the German DFG under contract HA 2966/1-1 and -2.

the logic CTL before presenting sequential model checking algorithms for CTL over PNs in Sect. 4. Section 5 then presents a number of distributed algorithms for the same purpose. Experimental results for two benchmark models are reported in Sect. 6. Finally, Sect. 7 presents related work, and Sect. 8 concludes the paper.

## 2 Petri net preliminaries

### *2.1 Stochastic Petri nets and Markov chains*

In this paper, we consider a simple class of (stochastic) Petri nets (PNs), including exponentially delayed transitions, inhibitor arcs, and marking-dependent rates; we do not consider marking-dependent arc multiplicities or immediate transitions. Notationally, we follow [21, Chap. 14].

The PNs we consider consist of a set of places $\mathcal{P} = \{\mathtt{P}_1, \cdots, \mathtt{P}_n\}$ (depicted as circles), a set of transitions $\mathcal{T} = \{\mathtt{T}_1, \cdots, \mathtt{T}_m\}$ (depicted as bars), a multiset of directed arcs $\mathcal{A} \subset (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P})$ (depicted as arrows between places and transitions or transitions and places), and a multiset of inhibitor arcs $\mathcal{H} = \{h_1, \cdots, h_l\}$, where $\mathcal{H} \subseteq \mathcal{P} \times \mathcal{T}$ (depicted as lines ending with a small circle, from places to transitions). Places may contain zero or more tokens, depicted as black dots (or a numerical value) in the places. The marking or state of a PN is the vector describing the distribution of tokens over places, denoted as $\underline{m}$ where $m_i$ denotes the number of tokens in place $\mathtt{P}_i$.[1]

The dynamics of a PN can be described as follows. A transition $t \in \mathcal{T}$ is called enabled when all its input places, that is, all places $p$ such that $(p,t) \in \mathcal{A}$, contain at least a token, and none of its inhibiting places, that is, places $p$ such that $(p,t) \in \mathcal{H}$, contains tokens. An enabled transition may fire, thus taking a token from all of its input places and depositing a token in all of its output places, that is, places $p$ such that $(t,p) \in \mathcal{A}$. Thus the firing of a transition can bring the PN into a new state (or marking). In the case of multiple arcs between a transition and its connected places, the corresponding number of tokens is consumed from or generated in the place.

Although not explicitly necessary for this paper, typical stochastic Petri nets (SPNs) used for performance evaluation purposes associate a negative exponentially distributed delay with the firing of a transition, given by a rate value (which is the reciprocal of the mean delay). In doing so, the SPN can be seen as a high-level description for a stochastic process, more specifically, for a continuous-time Markov chain (CTMC) with state space $S$, where every state $s \in S$ corresponds to a particular distribution of tokens over the places, i.e., a marking. Sometimes, next to exponentially distributed delays, zero-delay transitions, so-called immediate transitions, are also allowed, but in that case the underlying stochastic process is also a CTMC. The same may be said in cases where we allow the rates of the transitions to be marking dependent, or if we allow the multiplicity of the arcs or of the inhibitor arcs to be marking dependent; we do not consider these cases in this paper in order to avoid unnecessary complex notation. For seminal publications on this topic, refer to [1, 10].

In general, it is possible to specify with a finite SPN an underlying CTMC that has infinitely many states. The models addressed for performance evaluation purposes, however, typically result in finite-state CTMCs; in this paper we restrict ourselves to such models. A sufficient condition for such models is that every place in the SPN must be covered by a place invariant. There are, however, SPNs of which not all places are covered by a place invariant but that still have an underlying Markov chain that is finite. A general decidability result regarding the finiteness of the Markov chain underlying the SPN (which may include inhibitor arcs and timed as well as immediate transitions) is, however, unknown to us.[2]

To describe the PNs, we use the language CSPL, as defined by Ciardo et al. for the tool SPNP [10]. For the current paper, we do not use the stochastic properties of the PN being specified; this is left for future studies in which we will address distributed model checking of CSL or CSRL for SPNs.

### *2.2 State space and reachability graph generation*

Using a simple search algorithm, we can compute all reachable states (the state space $S$) as well as the transition relation (a subset of $S \times S$). Together these form the (directed) labeled reachability graph (RG). For that purpose, we have developed the PARSECS state space generator (**Par**allel **S**tate-space **E**xplorer and Markov **C**hain **S**olver); it exists in a serial and a distributed version and can generate state spaces with several hundred million states in a reasonable amount of time. PARSECS fully supports the PN language CSPL and is used as the basis for the algorithms described in the current paper. PARSECS has been written in C/C++ and uses the de facto standard MPI (message passing interface) [31] to facilitate communication in distributed computations; for more details, we refer the reader to [22]. In the rest of this paper we assume that the state space has been generated and is available in main memory.

As we will see later, the algorithms we develop require us to be able to search quickly for the existence of a state $s$. For this reason, we decided to use hash tables, which make it possible to find a state in constant time, as opposed to, for instance, binary trees, which have logarithmic search times. Thus, given the state space $S$,

[1] We use the terms state (denoted $s$) and marking (denoted $\underline{m}$) interchangeably; when referring to a marking, we emphasize the fact that it comprises the number of tokens in the places of the PN, hence the "vector notation" $\underline{m}$.

[2] In untimed PNs without inhibitor arcs, the situation is much clearer; cf. [18].

we store it in a hash table of size $N_{|S|} = c \cdot |S|$ for some $c > 1$. Notice that, given a PN, we do not know the size of the state space a priori; in practice, we therefore allow as much memory as is available on our machines for the hash table. The actual design of the hash table data structure (based on double hashing with open addressing; cf. [28, Sect. 6.4]) is not the topic of the current paper; it has been addressed in [4, 22]. Our experiments, however, show that we can check for the existence of a state using, on average, less than three `compare` operations for $c = 1.2$. In Sect. 6 we will elaborate in more detail on the influence of the parameter $c$. The hash-table-based representation has been shown to be far superior to representations based on tree structures [4].

The state spaces we can explicitly generate are limited by the amount of memory available to store them. The transition relation (that is, the so-called generator matrix of the CTMC) generally is much too large to maintain in main memory. Therefore, during state space generation it is written to disk in some sparse matrix format. To save storage, we decided not to use the transition relation directly in our model checking algorithms but instead to recompute successor and predecessor states whenever necessary, as will be outlined below. In this way we avoid costly disk accesses at the price of recomputations.

### *2.3 Successor and predecessor computation*

*Successor states.* The successor states of a given marking $\underline{m}$ can easily be determined from the semantics of the PN by considering the set of enabled transitions in that marking (determined by checking the enabling conditions of all transitions) and computing the new marking $\underline{m}'$ arising if in marking $\underline{m}$ a particular transition $t \in \mathcal{T}$ fires. The pseudocode for this algorithm is given in Algorithm 1.

In the algorithm, we use the following notation. Let $c_{t,p}$ be the net effect on the marking in place $p$ when transition $t$ fires. Clearly, we have $c_{t,p} = O(t,p) - I(t,p)$, where $O(t,p)$ is the number of tokens generated in $p$ when transition $t$ fires ("**o**utput arcs from $t$ to $p$") and $I(t,p)$ is the number of tokens consumed from $p$ when transition $t$ fires ("**i**nput arcs from $p$ to $t$") . The PN incidence matrix $\mathbf{C} = [c_{t,p}]$ summarizes these values. If $\underline{e}_t$ is a vector of zeroes with a single 1 at the $t$-th position, then the firing of $t$ changes a marking $\underline{m}$ in a new marking $\underline{m}' = \underline{m} + \mathbf{C} \cdot \underline{e}_t = \underline{m} + \underline{c}_t$, where $\underline{c}_t$ is the column in $\mathbf{C}$ associated with the firing of transition $t$.

**Algorithm 1 (Compute successors of marking $\underline{m}$: Succ $(\underline{m})$)**

0. given $\underline{m}$; Succ $(\underline{m}) \leftarrow \emptyset$;
1. for all $t \in \mathcal{T}$
2. do if Enabled $(t, \underline{m})$
3. then Succ $(\underline{m}) \leftarrow$ Succ $(\underline{m}) \cup \{\underline{m} + \underline{c}_t\}$;
4. od;
5. return Succ $(\underline{m})$;

*Predecessor states.* In many applications it is important to have a means of exploring the transition relation in the backward direction, that is, one wants to know all (or one) predecessor state(s) of a given state $\underline{m}$. This need occurs when solving Markov chains associated with an SPN in an iterative fashion [21] or when model checking `next` and `until` operations ([15, 26]; see below).

Consider a marking $\underline{m}$; it has been reached from another marking by the firing of at least one of the transitions $t \in \mathcal{T}$. Hence we have for the set of *possible predecessors* of $\underline{m}$: PosPred $(\underline{m}) = \{\underline{m} - \underline{c}_t | t \in \mathcal{T}\}$. This set is, in general, a superset of the set of real predecessors, for two reasons: (i) a state of the form "$\underline{m} - \underline{c}_t$" does not necessarily exist for all $t$ and given $\underline{m}$; (ii) if the state $\underline{m} - \underline{c}_t$ does exist, it might be the case that $t$ is not enabled in it, so that $\underline{m}$ cannot be reached from it. Hence we have to "shrink" PosPred $(\underline{m})$ accordingly, to yield Pred $(\underline{m})$, as shown in Algorithm 2.

**Algorithm 2 (Compute predecessors of marking $\underline{m}$: Pred $(\underline{m})$)**

0. given $\underline{m}$; Pred $(\underline{m}) \leftarrow \emptyset$;
1. for all $t \in \mathcal{T}$
2. do $\underline{m}' \leftarrow \underline{m} - \underline{c}_t$; /* $\underline{m}' \in$ PosPred $(\underline{m})$ ! */
3. if $\underline{m}' \in S$ /* lookup whether $\underline{m}'$ exists at all */
4. then if Enabled $(t, \underline{m}')$
5. then Pred $(\underline{m}) \leftarrow$ Pred $(\underline{m}) \cup \{\underline{m}'\}$;
6. od;
7. return Pred $(\underline{m})$;

Notice that this procedure is very similar to the one for finding successors. The only difference is that we first compute the possible predecessors, check their existence, and then check on their enabledness, whereas for finding successors, we first check on enabledness and then compute the successors. Note that, in contrast to the successor computation, we need to have knowledge of the complete state space $S$ when computing predecessors.

### *2.4 Simple state properties*

Instead of formally defining simple state properties, they are informally defined here as simple comparisons of numerical expressions involving place markings and constants. Since we use the language CSPL also for the simple state properties, the full flexibility of C is available for their specification (for this reason, we do not provide a syntax for simple state properties here). The only restriction we have to obey is that simple state properties can be evaluated to their truth value (`true` or `false`)

when *only the current marking (state)* is known. Hence, simple expressions do not require information about successor or predecessor states; their truth values can be seen as atomic propositions associated with each state.

As an example, let $\{\mathtt{P}_1, \mathtt{P}_2, \mathtt{P}_3, \mathtt{P}_4\}$ be the set of places in a PN and let $\#\mathtt{P}_i$ denote the number of tokens in place $\mathtt{P}_i$ in the current marking. Examples of simple state properties then are: $\#\mathtt{P}_1 < 3$, $\#\mathtt{P}_2 \geq (2 \times \#\mathtt{P}_4)$, and $(\#\mathtt{P}_1/\#\mathtt{P}_2) < 3 + 2 \times (\#\mathtt{P}_3 - \#\mathtt{P}_4)$.

In every state, a simple state property is either true or false. Therefore, with every state we associate a bit-vector equal in length to the number of simple state properties we consider. The bit values in this vector encode the truth value of the corresponding simple state properties.

We are aware of the fact that specialized data structures exist to encode especially small sets. Since some simple expressions might evaluate to true (or false) in only very few states, such data structures, e.g., based on binary decision diagrams, might be less memory consuming in such cases. A comparison of various data structures to present sets, however, goes beyond the scope of this paper.

## 3 Computational tree logic

### *3.1 Syntax*

In this paper we consider the following syntax for CTL (taken from [26, Chap. 3]). Let AP be the set of atomic properties (consisting of simple state properties as defined in Sect. 2.4), $p \in \mathrm{AP}$, then a CTL formula $\varphi$ is defined (in BNF) as

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{EX}\,\varphi \mid \mathsf{E}\,[\varphi\,\mathsf{U}\,\varphi] \mid \mathsf{A}\,[\varphi\,\mathsf{U}\,\varphi]\,.$$

Other boolean operators, such as true (`tt`), false (`ff`), and ($\wedge$) or implication ($\rightarrow$), are defined as usual. Note that the above definition allows for so-called nested CTL expressions, that is, expressions in which the `next` ($\mathsf{EX}\,\varphi$) and `until` operators ($\mathsf{E}\,[\varphi\,\mathsf{U}\,\varphi]$ or $\mathsf{A}\,[\varphi\,\mathsf{U}\,\varphi]$) appear as subformulas of such expressions.

### *3.2 Semantics*

The models we will check CTL formula for are models defined as high-level PNs, that is, the set of states is the set of reachable markings $S$, the transition relation $R \subseteq S \times S$ is defined by the firing semantics of the PN and corresponds to the (labeled) edges in the reachability graph. Furthermore, the atomic propositions associated with each state indicate whether the simple expressions hold in each state. Finally, we define a path $s^{(0)}, s^{(1)}, s^{(2)}, \cdots$ as a sequence of states such that in state $s^{(i)}$ a transition $t \in \mathcal{T}$ is enabled and upon firing yields state $s^{(i+1)}$.

A model checking algorithm now verifies the following satisfaction relation $\models$ (for $p \in \mathrm{AP}$, $\varphi$, $\varphi_1$, and $\varphi_2$ CTL formula, $s$ a state):

$$\begin{array}{ll}
s \models p & \text{iff } p \text{ evaluates to } \mathtt{true} \text{ in } s \\
s \models \neg\varphi & \text{iff not } (s \models \varphi) \\
s \models \varphi_1 \vee \varphi_2 & \text{iff } (s \models \varphi_1) \text{ or } (s \models \varphi_2) \\
s \models \mathsf{EX}\,\varphi & \text{iff } s' \models \varphi \text{ for some } (s, s') \in R \\
s \models \mathsf{E}\,[\varphi_1\,\mathsf{U}\,\varphi_2] & \text{iff for } \textit{some} \text{ path } (s^{(0)}, s^{(1)}, s^{(2)}, \cdots), \\
 & \quad \exists i \geq 0 : [s^{(i)} \models \varphi_2 \wedge \\
 & \qquad (\forall j, 0 \leq j < i : s^{(j)} \models \varphi_1)] \\
s \models \mathsf{A}\,[\varphi_1\,\mathsf{U}\,\varphi_2] & \text{iff for } \textit{all} \text{ paths } (s^{(0)}, s^{(1)}, s^{(2)}, \cdots), \\
 & \quad \exists i \geq 0 : [s^{(i)} \models \varphi_2 \wedge \\
 & \qquad \forall j, 0 \leq j < i : s^{(j)} \models \varphi_1)]
\end{array}$$

## 4 CTL model checking of Petri nets

### *4.1 Decomposing CTL formulas*

A model checking procedure verifies the validity of properties in a recursive manner. To be able to model check nested CTL formulas, we have to distinguish the subformulas in these formulas. Subformulas of length 1 correspond to atomic properties (or simple state properties, in the sense of Sect. 2.4), of which the validity can be directly established (see below). Based on their validity, the validity of subformulas involving simple logical operators or the `next` and `until` operators can be established using the algorithms described below. The *length* of a CTL formula is defined as the number of subformulas in the CTL formula. The complexity of the model checking procedure is linear in this length. Clarke et al. describe this recursive approach using the notion of a parse tree of a CTL formula [15]; as an example, a parse tree for the CTL formula $\mathsf{A}\,[\neg X\,\mathsf{U}\,(Y \vee Z)]$ is given in Fig. 1 (with $X$, $Y$, and $Z$ atomic properties).

### *4.2 Model checking at state space generation time*

We associate with every state in the considered PN an array of booleans, with length equal to the number of subformulas in the CTL expression to be checked. The $i$-th bit in this array indicates whether the $i$-th subformula evaluates to true or false.

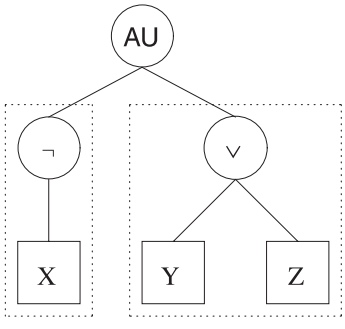



**Fig. 1.** Parse tree for the simple CTL formula $\mathsf{A}\,[\neg X\,\mathsf{U}\,(Y \vee Z)]$

The truth values of the subformulas encompassing only simple expressions can already be evaluated at state space generation time. Notice that, in fact, it suffices to store only the truth value of complete subformulas not involving the `next` and `until` operators. As a further enhancement one could even eliminate common subformulas. Neither of these optimizations, however, is further addressed in this paper. Hence, the only "problem" remaining is the evaluation of subformulas in which the `next` and `until` operator(s) occur.

### *4.3 Model checking propositional logic expressions*

Let $\varphi_1, \varphi_2$ be either simple expressions or results from previous model checking steps. In each case the sets of states satisfying $\varphi_1, \varphi_2$, denoted as $\mathrm{Sat}(\varphi_1)$ and $\mathrm{Sat}(\varphi_2)$ respectively, are known and represented as bit-vectors, as described in Sects. 2.4 and 4.2. Model checking $\varphi = \neg\varphi_1$ is performed by setting $\mathrm{Sat}(\varphi) \leftarrow \overline{\mathrm{Sat}(\varphi_1)}$, which can be implemented by negating every bit of the corresponding bit-vector. Model checking a formula of the type $\varphi = \varphi_1 \wedge \varphi_2$ is performed by setting $\mathrm{Sat}(\varphi) \leftarrow \mathrm{Sat}(\varphi_1) \cap \mathrm{Sat}(\varphi_2)$, which can be implemented as a logical *AND* between two bits of the corresponding bit-vectors for $\varphi_1$ and $\varphi_2$. The operators $\vee$, $\rightarrow$, $\leftrightarrow$, etc. can be checked in a similar way. Notice that if $\varphi_1$ and $\varphi_2$ do not contain subformulas involving the `next` and `until` operators, the approach just sketched can be performed already at state space generation time (Sect. 4.2).

### *4.4 Model checking $\varphi = \mathsf{EX}\varphi_1$*

Now, consider the case where we have to find those states that satisfy $\varphi = \mathsf{EX}\varphi_1$. Again we assume that the states satisfying $\varphi_1$ ($\mathrm{Sat}(\varphi_1)$) are known. We can then proceed with either a forward or a backward search.

Let us first address the forward search case, as shown in Algorithm 3. The set $\mathrm{Sat}(\varphi)$ (states satisfying $\varphi$) is initially empty. Then, for each of the states $s \in S$, we compute the set of successor states $s'$, denoted by $\mathrm{Succ}(s)$. As soon as we stumble upon a state $s'$ in which $\varphi_1$ holds, which can be directly seen from its associated bit-vector, we add $s$ to $\mathrm{Sat}(\varphi)$ and exit the inner loop ranging from line 2 to line 5.

**Algorithm 3 (Forward computation of $\varphi = \mathsf{EX}\varphi_1$)**

```
0. Sat(φ) ← ∅; /* Sat(φ1) is known */
1. for all s ∈ S
2. do for all s' ∈ Succ(s)
3.      do if s' ∈ Sat(φ1)
4.           then Sat(φ) ← Sat(φ) ∪ {s}; break;
5.      od;
6. od;
7. return Sat(φ);
```

In the backward search case, we use the predecessor function instead of the successor function, as shown in Algorithm 4.

**Algorithm 4 (Backward computation of $\varphi = \mathsf{EX}\varphi_1$)**

```
0. Sat(φ) ← ∅; /* Sat(φ1) is known */
1. for all s' ∈ Sat(φ1)
2. do for all s ∈ Pred(s')
3.      do
4.           Sat(φ) ← Sat(φ) ∪ {s};
5.      od;
6. od; /* exit outer loop whenever Sat(φ) = S */
7. return Sat(φ);
```

The forward variant requires a single iteration through the complete state space $S$. The backward computation appears to be more efficient since it only requires a "for all" clause over states in $\mathrm{Sat}(\varphi_1)$; *however*, given the data structure we use, this clause has to be implemented as "for all $s' \in S$ if $s' \in \mathrm{Sat}(\varphi_1)$". Thus, although $S$ is generally larger than $\mathrm{Sat}(\varphi_1)$, in both cases we need to go through the entire hash table and check, for every entry, whether it represents a state at all.[3] In any case, each hash table entry contains enough information on the actual marking it represents that we easily establish its predecessors and, hence, set the corresponding bits (for $\mathsf{EX}\varphi_1$) in the satisfying predecessor states (in the bit-vector representing the result $\mathrm{Sat}(\varphi)$).

### *4.5 Model checking $\varphi = \mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$*

Here we only consider the backward variant and assume, as before, that $\mathrm{Sat}(\varphi_1)$ and $\mathrm{Sat}(\varphi_2)$ are known. We construct a sequence of sets $\mathrm{Sat}^{(0)}(\varphi) \subseteq \mathrm{Sat}^{(1)}(\varphi) \subseteq \mathrm{Sat}^{(2)}(\varphi) \subseteq \cdots$ until two successive elements of this sequence are identical. Initially, we set $\mathrm{Sat}^{(0)}(\varphi) = \mathrm{Sat}(\varphi_2)$ since states satisfying $\varphi_2$ automatically satisfy $\varphi$. We denote by $\mathrm{Sat}'(\varphi) = \mathrm{Sat}^{(k+1)}(\varphi) \setminus \mathrm{Sat}^{(k)}(\varphi)$ (for $k \geq 0$) the set of states that was found to satisfy $\varphi$ in the last step in the iterative procedure and was not already in $\mathrm{Sat}^{(k)}(\varphi)$. These are the states for which we have to check whether their predecessors satisfy $\varphi_1$.

The pseudocode for this algorithm is shown as Algorithm 5. Initially, $\mathrm{Sat}'(\varphi) = \mathrm{Sat}(\varphi_2)$, as we have not looked at paths leading to the elements in $\mathrm{Sat}(\varphi_2)$. The set $S_{\mathrm{new}}$ is used to store the states we insert into $\mathrm{Sat}(\varphi)$ during the $k$-th iteration, giving us the set $\mathrm{Sat}'(\varphi)$ for the next iteration. In each iteration we then enlarge the set $\mathrm{Sat}(\varphi)$ by including the predecessor states that we do not already know in which $\varphi_1$ holds until we do not find any new state(s) ($\mathrm{Sat}'(\varphi) = \mathrm{Sat}^{(k+1)}(\varphi) \setminus \mathrm{Sat}^{(k)}(\varphi) = \emptyset$).

[3] For each hash table entry we know whether it is empty, meaning it does not represent a state, or not.

**Algorithm 5 (Backward computation of $\varphi = \mathsf{E}\,[\varphi_1\,\mathsf{U}\,\varphi_2]$)**

```
0.  Sat (φ) ← Sat (φ2); Sat′ (φ) ← Sat (φ2); S_new ← ∅
1.  while Sat′ (φ) ≠ ∅
2.      do for all s ∈ Sat′ (φ)
3.          do for all s′ ∈ Pred (s)
4.              do if ((s′ ∉ Sat (φ)) ∧ (s′ ∈ Sat (φ1)))
5.                  then
6.                      S_new ← S_new ∪ {s′};
7.                      Sat (φ) ← Sat (φ) ∪ {s′};
8.              fi; od;
9.          od;
10.     Sat′ (φ) ← S_new; S_new ← ∅;
11.     od;
12. od;
13. return Sat (φ);
```

We do not check for emptiness explicitly (in step 1) but keep track of the number of elements in the set. Notice that we add those predecessor states $s$ (in steps 3–7) that have *at least one* successor for which the required property holds. We do *not* require the looked-after property to hold for *all* successors of the added states. This fact gives rise to the simple inclusion in the set $\mathrm{Sat}\,(\varphi)$. We can first generate all predecessors (step 3); we are automatically sure that these have at least one successor in which the required property holds. We then select the ones (individually) in which $\varphi_1$ holds and that are not already an element of $\mathrm{Sat}\,(\varphi)$ before we join all these to the ones we already had.

At the implementation level there exists an optimization for this algorithm that decreases the required number of iterations until a fix point is reached. The "for all" clause in step 2 is implemented as a loop over all entries of the hash table. Assume that the currently inspected element $s$ corresponds to entry $j$ in the hash table. If we then find a predecessor $s'$ that satisfies the if clause in step 4 and of which the index in the hash table is $l > j$, then we can insert it into $\mathrm{Sat}'\,(\varphi)$ (by setting the appropriate bit to `true`) instead of $S_{\mathrm{new}}$ and its predecessors will already be inspected in the current iteration. Results for the simple and the optimized implementation of this algorithm are given in Sect. 6.

### *4.6 Model checking $\varphi = \mathsf{A}\,[\varphi_1\,\mathsf{U}\,\varphi_2]$*

For a model checking formula of the form $\varphi = \mathsf{A}\,[\varphi_1\,\mathsf{U}\,\varphi_2]$, we can proceed similarly as in the $\mathsf{E}\,[\varphi_1\,\mathsf{U}\,\varphi_2]$ model checking algorithm by constructing a sequence of sets $\mathrm{Sat}^{(0)}(\varphi) \subseteq \mathrm{Sat}^{(1)}(\varphi) \subseteq \mathrm{Sat}^{(2)}(\varphi) \subseteq \cdots$ until two successive elements of this sequence are the same. Notice that it does not suffice to consider only the states newly generated during the last iteration; we have to reconsider each state again in each iteration. Furthermore, we have to make sure that, in contrast to steps 3–6 of Algorithm 5, here only those states are added from which *all* successor states lead to states that are already known to satisfy the property. The pseudocode of this is given in Algorithm 6.

**Algorithm 6 (Backward computation of $\varphi = \mathsf{A}\,[\varphi_1\,\mathsf{U}\,\varphi_2]$)**

```
0.  Sat′ (φ) ← ∅; Sat (φ) ← Sat (φ2);
1.  while Sat′ (φ) ≠ Sat (φ)
2.      do Sat′ (φ) ← Sat (φ);
3.      for all s ∈ Sat (φ)
4.          do for all s′ ∈ Pred (s) ∩ ¬Sat (φ) ∩ Sat (φ1)
5.              if (Succ (s′) ⊆ Sat (φ))
6.              then Sat (φ) ← Sat (φ) ∪ {s′};
7.          fi; od;
8.      od;
9.  return Sat (φ);
```

The check for equality of the sets $\mathrm{Sat}'\,(\varphi)$ and $\mathrm{Sat}\,(\varphi)$ in step 1 does not need to be done in an elementwise manner. We can just look at the number of elements to check whether $\mathrm{Sat}\,(\varphi)$ grew during the last iteration. Before checking that all successors of $s'$ satisfy $\varphi$, which is an expensive test, we make sure that $s'$ is not already in $\mathrm{Sat}\,(\varphi)$ and satisfies $\varphi_1$. One can see why we have to reconsider states in step 5: whenever we have new elements in $\mathrm{Sat}\,(\varphi)$, the result of this test may change.

## 5 Distributed implementation

### *5.1 Introduction*

In the distributed version of the state space generation, as described in detail in [22], every state is allocated to a particular processor using a hashing procedure; this hashing function is denoted as $A : S \to \{1, \cdots, NoP\}$ ("$A$" for allocation) and provides, for every state $s$, a unique processor allocation (assuming that the **n**umber **o**f **p**rocessors is $NoP$). In doing so, we store the state space as a partition: $S = \cup_{i=1}^{NoP} S_i$. By choosing a hashing function that allocates the states equally among the partitions $S_i$ we simultaneously distribute the work to be done evenly. The hashing functions we use typically yield partitions where the difference between the smallest and the largest partition is below 5%. Allocation functions yielding more uniform partition sizes are known but lead to more cross arcs, i.e., situations in which the successor $s' = \mathrm{Succ}\,(s)$ of a certain state $s$ does not belong to the same partition as $s$, that is, $A(s') \neq A(s)$. Larger numbers of cross arcs obviously require more communication during both state space generation and model checking. In the chosen examples (Sect. 6), the fraction of cross arcs was always below 30%. The attained speedups demonstrate that no further load balancing strategies are necessary. For more details on state space partitioning see [4, 22].

With respect to the size of the hash table, the conditions for the serial case now hold for each individual processor. Within each processor, a hashing table is used to store the assigned states (and all other required information), as in the sequential case.

Whenever we send states from one processor to another, we use buffers. Every processor has an output buffer (sized between 2 and 4 KB for fast Ethernet) for every other processor. Smaller buffers lead to an inefficient usage of the available bandwidth, whereas larger buffers have higher memory requirements and bring no additional advantages. A buffer is flushed, i.e., all the states it stores are sent to the receiving processor, if it is either full or a certain timeout value (typically 0.5–2.0 s) has been reached. A low timeout value assures that a computation gets started (and keeps running) at all. Consider the scenario in which the processor "owning" the initial state generates only states for which other processors are responsible; the faster the buffers are flushed in this situation, the earlier the actual computation starts to operate on all processors. On the other hand, if this timeout value is chosen too small, the buffers will never fill completely and the usage of the available bandwidth will be unsatisfactory. Obviously both parameters, buffer size and timeout value, depend on the employed hardware; in our previous work we conducted a variety of experiments to come up with the values indicated above, for which the observed performance is satisfactory.

Finally, in our algorithms we use `send` and `receive` operations as made available via the MPI library. All receive calls have been used in a nonblocking fashion. In particular, we use nonblocking MPI functions to periodically check whether messages arrived and call receive functions only if there are messages in the receive buffers. In doing so, we eliminate the possibility of deadlocks.

### *5.2 Model checking propositional logic expressions*

In the distributed implementation, model checking simple expressions and propositional logic expressions does not cause any problem since no communication is required at all for this purpose. All processors just work on their partitions of the state space as in the serial case. As expected, this results in speedups that are only limited by the load balance induced by the allocation function.

### *5.3 Model checking $\varphi = \mathsf{EX}\,\varphi_1$*

For the `next` operator we require a single pass over the complete state space, where we use the backward procedure. All *NoP* processors can, in principle, work in parallel here.

However, we cannot compute the set of predecessors of a given marking $\underline{m}$ locally since a predecessor of $\underline{m}$ need not be allocated on the same processor as $\underline{m}$. Hence, we have to compute the set $\mathrm{PosPred}(\underline{m})$, which might include nonexistent states, as given in Algorithm 7. For a given marking $\underline{m}$, this algorithm iterates over all transitions (line 1) and fires each transition backwards (line 2), resulting in a potentially existing marking $\underline{m}'$, which is added to the set of possible predecessors in line 3.

In the tool implementation, we shrink this set by removing local states that do not exist as well as states whose nonexistence can be deduced from invariants or known minimum/maximum number of tokens per place.

In our ongoing work, we investigated an alternative solution to check whether a potential state really exists, namely, by using an *additional* implicit representation of the state space on each processor.[4] This leads to a situation in which each processor can locally decide on the existence of every potential state. Binary and multivalued decision diagram (MDD) representations both meet the demand of having very low memory requirements to store complete state spaces and admit relatively fast lookups (although slower than hash table lookups) [25]; initial experiments show that the memory requirements for an additional MDD-based representation of the complete state space are negligible.[5] Using this approach, no nonexisting states have to be communicated; we have not yet performed extensive experiments with this approach.

**Algorithm 7 (Computation of $\mathrm{PosPred}(\underline{m})$)**

0. given $\underline{m}$; $\mathrm{PosPred}(\underline{m}) \leftarrow \emptyset$;
1. for all $t \in \mathcal{T}$
2. do $\underline{m}' \leftarrow \underline{m} - \underline{c}_t$;
3. $\mathrm{PosPred}(\underline{m}) \leftarrow \mathrm{PosPred}(\underline{m}) \cup \{\underline{m}'\}$;
4. od;
5. return $\mathrm{PosPred}(\underline{m})$;

Using the set of possible predecessors, the distributed (backward) computation of $\varphi = \mathsf{EX}\,\varphi_1$ for processor $i$ is given in Algorithm 8. As in the serial case, the outermost loop (lines 1–7) iterates over all states satisfying $\varphi_1$, with the difference that processor $i$ only considers states belonging to its subset $S_i$ of the state space.

**Algorithm 8 (Distributed computation of $\varphi = \mathsf{EX}\,\varphi_1$ for proc. $i$)**

0. $\mathrm{Sat}_i(\varphi) \leftarrow \emptyset$; $\mathrm{Sat}_i(\varphi_1)$ is known;
1. for all $s' \in \mathrm{Sat}_i(\varphi_1)$
2. do for all $s \in \mathrm{PosPred}(s')$
3. do if $(A(s) = i)$ AND $(s \in S_i)$ then
4. $\mathrm{Sat}_i(\varphi) \leftarrow \mathrm{Sat}_i(\varphi) \cup \{s\}$;
5. else
6. send$(A(s'), s', \texttt{msg_checkEX})$;
7. od;

[4] In Sect. 1 we argued why we want to have an explicit representation in the first place.

[5] Even the largest models we are able to analyze in parallel using 13 GB of aggregated RAM (around 750 million states) can be stored as an MDD using less than 100 KB of memory.

```
8.      od; /* exit loop whenever Sat_i(φ) = S_i */
9.  return Sat(φ);

    handle received states concurrently:

10. for all s ∈ {received states}
11.     do if s ∈ S_i then
12.         Sat_i(φ) ← Sat_i(φ) ∪ {s}
13.     od;
```

The loop (lines 2–8) iterates over all possible predecessors. Lines 3–4 add local ($A(s) = i$) states that really exist to the set $\mathrm{Sat}_i(\varphi)$, whereas the remaining elements $s \in \mathrm{PosPred}(s')$ that cannot be handled locally are sent, buffered as described before, to the processor $A(s)$ that is responsible for them (line 6).

Each processor periodically checks for received messages, either at some fixed point during the algorithm or at certain time intervals, or possibly even in another thread. The code that is executed for states received during the distributed computation of $\varphi = \mathsf{EX}\,\varphi_1$ is shown as lines 10–13 (still in Algorithm 8). For all states $s$ processor $i$ receives (line 10), it checks whether this state exists (line 11), that is, if there is an entry for $s$ in the local hash table. If this test returns `true`, the corresponding state $s$ is added to the local result $\mathrm{Sat}_i(\varphi)$ (line 12).

Note that there is no need for processor $A(s)$ to reply in this case. Processor $A(s)$ just checks whether or not a state $s$ exists. In the first case, $s$ is added to the local result; in the latter case, $s$ is just ignored. Hence, all processors just have to consider their local sets of states corresponding to their part of $\mathrm{Sat}(\varphi)$, as well as the states they receive from other processors. As soon as all processors have processed their local sets $\mathrm{Sat}(\varphi)$ and their receive buffers with states received from other processors are empty, the procedure ends.

To avoid useless communications, a processor receiving a potential state from a sending processor that finds that the received state does not exist could inform the sending processor of this fact. If that particular state then recurs in the rest of the model checking procedure, it could be filtered locally. However, doing this requires the storage of extra information at each processor as well as extra communication. Since these extra facilities pay off only when a potential state appears at least twice at a given processor, we expect the overhead to be larger than the savings; we therefore do not consider this further.

### *5.4 Model checking* $\varphi = \mathsf{E}\,[\varphi_1\,\mathsf{U}\,\varphi_2]$

In evaluating an `exist-until` formula, all processors again operate on their own part of the state space; the algorithm run by processor $i$ is given as Algorithm 9. The main differences compared to the serial case (see Algorithm 5) are that we can no longer check for the termination of the algorithm by testing the local set $\mathrm{Sat}'_i(\varphi)$ for emptiness but have to use a distributed ring check (line 1) to do so. Furthermore, processor $i$ cannot check whether a state $s$ belongs to $\mathrm{Sat}_j(\varphi_1)$, for $j \neq i$. The distributed ring check assures that all $\mathrm{Sat}'_i(\varphi)$ are empty and no messages are still in transit.

**Algorithm 9 (Distributed computation of $\varphi = \mathsf{E}\,[\varphi_1\,\mathsf{U}\,\varphi_2]$)**

```
0.  Sat_i(φ) ← Sat_i(φ_2); Sat'_i(φ) ← Sat_i(φ_2); S_new ← ∅
1.  while (not finished)   /* distributed ring check */
2.      do for all s ∈ Sat'_i(φ)
3.          do for all s' ∈ PosPred(s)
4.              do if (A(s) = i) then
5.                  if ((s' ∉ Sat_i(φ)) ∧ (s' ∈ Sat_i(φ_1)))
6.                      then
7.                          S_new ← S_new ∪ {s'};
8.                          Sat_i(φ) ← Sat_i(φ) ∪ {s'};
9.                  fi;
10.             else
11.                 send(A(s'), s', msg_checkEU);
12.             fi; od;
13.         od;
14.         Sat'_i(φ) ← S_new; S_new ← ∅;
15.     od;
16. return Sat_i(φ);

    handle received states concurrently:

17. for all s ∈ {received states}
18.     do if ((s ∉ Sat_i(φ)) ∧ (s ∈ Sat_i(φ_1))) then
19.         S_new ← S_new ∪ {s};
20.         Sat_i(φ) ← Sat_i(φ) ∪ {s};
21.     od;
```

A processor $i$ iterates over the set $\mathrm{Sat}_i(\varphi)$ until this set no longer changes. Note that the set $S_{\mathrm{new}}$ is a local variable available to each single processor. When determining predecessor states, the same can happen as for a `next` formula. Local predecessors are treated in the same way as in the serial algorithm (lines 4–9), but a computed possible predecessor $s' \in \mathrm{PosPred}(s)$ (line 3) need not necessarily be handled by the current processor; moreover, if $A(s) \neq A(s')$, processor $A(s)$ cannot even decide whether or not the computed state exists. Therefore, it sends state $s'$ to $A(s')$ (line 11), which then decides on existence. Note that changes to $\mathrm{Sat}_i(\varphi)$ can originate from the processor itself, as well as from other processors (see below). If for all processors the receive buffers are empty and the locally generated predecessors have been accounted for, the procedure ends.

Also in this case, received messages (states) have to be treated periodically. The corresponding code is given as lines 17–21. If the state exists and it is a member of $\mathrm{Sat}_i(\varphi_1)$, it has to be added to the set of unexplored states $S_{\mathrm{new}}$ and the result $\mathrm{Sat}_i(\varphi)$.

A similar implementation optimization as reported in the context of Algorithm 5 can be applied to Algorithm 9 as well.

### *5.5 Model checking $\varphi = \mathsf{A}\,[\varphi_1\,\mathsf{U}\,\varphi_2]$*

The distributed for-all-until check is also based on the serial version. In principle, we use Algorithm 6, in which every processor operates on its own partition $S_i$ of the state space and has its own local sets $\mathrm{Sat}_i(\varphi), \mathrm{Sat}'_i(\varphi), \mathrm{Sat}_i(\varphi_1)$, and $\mathrm{Sat}_i(\varphi_2)$, as illustrated in Algorithm 10. The check for termination is done by a distributed ring check in line 1. The main changes for the distributed algorithm are located in lines 4–14, replacing lines 4–6 of the serial algorithm.

**Algorithm 10 (Distributed computation of $\varphi = \mathsf{A}\,[\varphi_1\,\mathsf{U}\,\varphi_2]$ for proc. $i$)**

```
0.  Sat'_i(φ) ← ∅; Sat_i(φ) ← Sat_i(φ2);
1.  while (not finished)      /* distributed ring check */
2.      do Sat'_i(φ) ← Sat_i(φ);
3.      for all s ∈ Sat_i(φ)
4.          for all s' ∈ PosPred(s)
5.              do if A(s') = i then   /* a local state */
6.                  if (s' ∉ Sat_i(φ) ∧ s' ∈ Sat_i(φ1)) then
7.                      if local_Succ(s') ⊆ Sat_i(φ) then
8.                          if remote_Succ(s') = ∅ then
9.                              Sat_i(φ) ← Sat_i(φ) ∪ {s'};
10.                         else
11.                             Distr_Succ_Check(s');
12.             else     /* s' is not a local state */
13.                 send(A(s'), s', msg_checkAU);
14.             od;
15.         fi; od;
16.     od;
17. return Sat_i(φ);
```

As noted before, we cannot compute the predecessors of a state locally, so we have to iterate (line 4) over the possible predecessors of state $s$ (with $A(s') = i$). If the predecessor is a local state, i.e., $A(s') = i$, we check whether we do not already know it and whether it satisfies $\varphi_1$ (line 6). If all these tests hold true, we have to check whether all successors of $s'$ belong to $\mathrm{Sat}(\varphi)$. We do this in two steps. First, we check whether all local successors belong to $\mathrm{Sat}_i(\varphi)$ (line 7), and only if this is true do we check whether there are any remote successors (line 8). If there are no remote successors, we can add $s'$ to $\mathrm{Sat}_i(\varphi)$ (line 9), else we do a distributed successor check (line 11), which we describe below. Note that we do this distributed check only after we test everything that we can inspect locally. If $s'$ is not a local state, we cannot even check whether this state exists, so we send it to the corresponding processor (line 13) with the message identifier `msg_checkAU`.

Each processor periodically checks for received messages. States received with the identifier `msg_checkAU` are handled by the Recv_msg_checkAU procedure given in Algorithm 11. This is essentially the same as the handling of local states, but we have to check whether the received state exists (line 2) instead of checking whether it is a local state.

**Algorithm 11 (Handle received states Recv_msg_checkAU())**

```
1. for all s' ∈ {received states}
2.   if s' ∈ S_i then     /* does this state exist? */
3.       if (s' ∉ Sat_i(φ) ∧ s' ∈ Sat_i(φ1))
4.           then if local_Succ(s') ⊆ Sat_i(φ)
5.                then if remote_Succ(s') = ∅
6.                     then Sat_i(φ) ← Sat_i(φ) ∪ {s'};
7.                else Distr_Succ_Check(s');
```

A further improvement, as suggested by one of the reviewers, might be to include in the message sent in line 13 the information that $s$ is the successor state satisfying $\mathsf{A}\,[\varphi_1\,\mathsf{U}\,\varphi_2]$. When processor $A(s')$ stores this information, this might reduce the number of requests from $A(s')$ to processor $A(s)$ in the future. The question whether the overhead (extra data structure plus local check) is worth the potential gain is left for future research.

So far, the distributed algorithm for $\varphi = \mathsf{A}\,[\varphi_1\,\mathsf{U}\,\varphi_2]$ appears no more complicated than the algorithm for $\varphi = \mathsf{E}\,[\varphi_1\,\mathsf{U}\,\varphi_2]$, but we have not yet considered the distributed successor check. For this we cannot just send states (messages) and forget about them. To implement the distributed check, processor $i$ *asks* each processor $j$ responsible for some successors the *question* whether all these successors are elements of $\mathrm{Sat}_j(\varphi)$. Processor $j$ has to answer this question in each case (true or false). When processor $i$ receives `true` as answer, it can remove the corresponding question from the local list of unanswered questions. When removing the last open question of a specific state, we know that we received only `true`s and by this that we can add $s'$ to $\mathrm{Sat}_i(\varphi)$. If we receive a `false`, all questions regarding the original state $s'$ can be removed from the list. If an answer corresponding to a state for which no question exists is received, we can just drop it, as processor $i$ must have received a `false` answer before.

Finally, notice that the distribute ring check for node $i$ in Algorithm 10 can signal "work done" when in the last iteration no new states have been added to $\mathrm{Sat}_i(\varphi)$ (line 9) *and* there are no open questions anymore.

## 6 Experimental work

### *6.1 Distributed computing platform*

We ran all our experiments on the cluster in the Computer Science Department at the RWTH Aachen, consisting of 26 dual Pentium III (500 MHz) Linux workstations, each equipped with 512 MB main memory and

**Fig. 2.** Cluster in Computer Science Department at RWTH Aachen

40 GB local disk connected via switched fast Ethernet (100 Mbps). The cluster is shown in Fig. 2.

### *6.2 Cases addressed*

We addressed two case studies: a PN model from a kanban production system that is widely used as a benchmark in the performance evaluation community [9] and a PN representing the well-known dining-philosopher problem [12, 13]. Both models have in common that they can be easily scaled to larger state spaces by just changing one of the model parameters.

#### 6.2.1 Kanban model

The graphical PN representation of the kanban model is shown in Fig. 3. The model presents an abstract view of four connected production lines. The first production

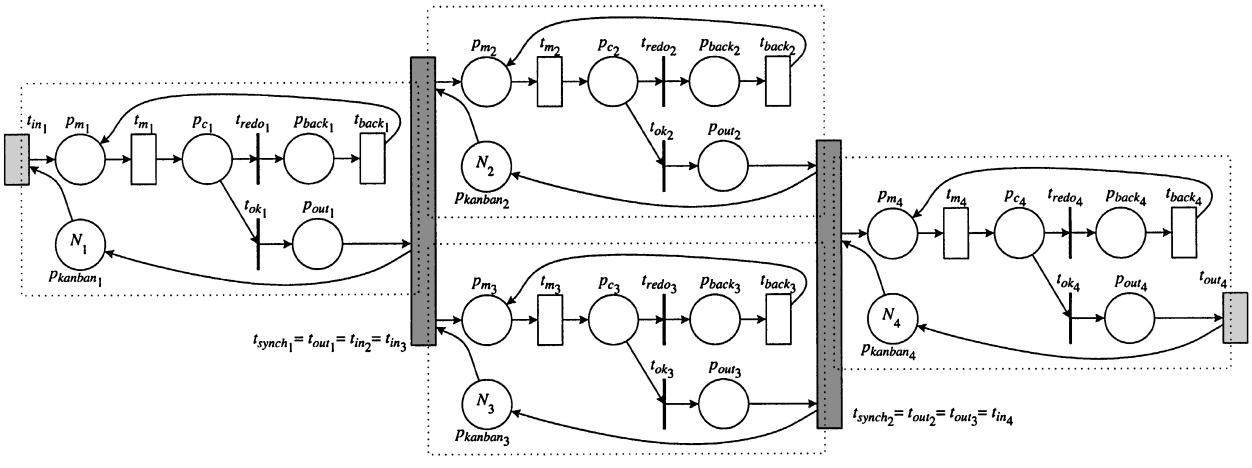



**Fig. 3.** Kanban model

**Table 1.** Statistics and serial tests (h:m:s.s) for kanban model

| $N$ | States | Arcs | Gen. $S$ | Read $S$ | Memory | AP | $\varphi_1 \wedge \varphi_2$ |
|---|---|---|---|---|---|---|---|
| 1 | 160 | 616 | 0:00.4 | 0:00.2 | $<1$ MB | 0:00.0 | 0:00.0 |
| 2 | 4600 | 28 120 | 0:00.5 | 0:00.2 | $<1$ MB | 0:00.0 | 0:00.0 |
| 3 | 58 400 | 446 400 | 0:02.5 | 0:00.4 | 1.5 MB | 0:00.1 | 0:00.0 |
| 4 | 454 475 | 3 979 850 | 0:21.1 | 0:00.6 | 4.5 MB | 0:01.2 | 0:00.0 |
| 5 | 2 546 432 | 24 460 016 | 2:26.8 | 0:02.8 | 20 MB | 0:06.5 | 0:00.2 |
| 6 | 11 261 376 | 115 708 992 | 14:22.9 | 0:14.0 | 89 MB | 0:29.2 | 0:00.7 |
| 7 | 41 644 800 | 450 455 040 | 51:15.0 | 0:50.7 | 326 MB | 1:48.0 | 0:02.7 |
| 8 | 133 865 325 | 1 507 898 700 | – | – | – | – | – |
| 9 | 383 933 678 | 4 176 462 582 | – | – | – | – | – |

line (leftmost model part) produces half-products that are buffered for further processing. The middle two production lines take products from that buffer and produce enhanced products out of them, which are again buffered. The rightmost model part presents a production line that uses two half-products as input to produce an end product.

The kanban model is parameterized with the number of tokens $N$ in each of the four subsystems in the initial marking (in the places $P_{kanban_i}$, for $i = 1, \ldots, 4$). The model exists in two variants, one with timed and one with immediate synchronizing transitions ($t_{\text{sync}_1}$ and $t_{\text{sync}_2}$). We only used the variant with timed synchronization. Table 1 shows the number of states and arcs (transitions in the Markov chain) for given $N$ in the first three columns. The hash tables used to store the state spaces were sized 20% larger than the size of the state space ($N_{|S|} = 1.2 \cdot |S|$). We were able to do so, as we knew the number of states from the state space generation step; if we do not know the number of states in advance, we allocate a hash table as large as our main memory allows. Note that the cases $N = 8, 9$ cannot be handled serially on a single node.

### 6.2.2 Dining philosophers

The dining-philosophers model (in different variants) has been used in a variety of case studies to test reachability and model checking algorithms. We use the variant described in [12, 13],[6] for which the PN model for the $i$-th philosopher is depicted in Fig. 4. In the $N$-philosopher case, $N$ of these models are connected to each other in a cyclic manner in such a way that the place for the left fork of philosopher $i$ ($\text{Fork}_{(i+1) \bmod N}$) coincides with the right fork of his left neighbor (philosopher $(i+1) \bmod N$). Notice that this model does contain deadlock states, e.g., when each philosopher picks up his right fork. The state space of this model is parametric in the number of philosophers; the number of states and arcs as well as the state space generation time (in case of serial algorithms) are given in Table 2.

[6] The technical report does contain a detailed description and a picture of the model; the LNCS paper contains just test results related to the model.

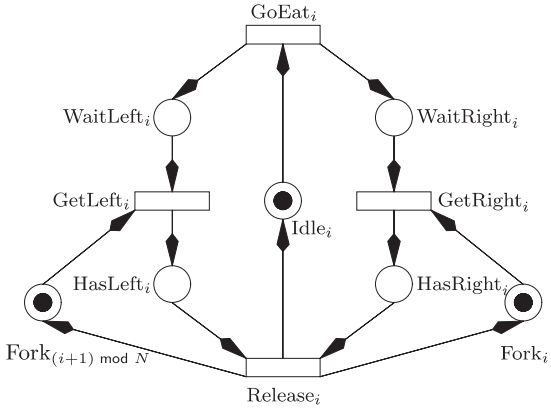



**Fig. 4.** Dining philosopher $i$ out of $N$

**Table 2.** State space size, number of arcs, and (wall clock) generation time for the dining-philosophers model with $N$ philosophers

| $N$ | Time | States | Arcs |
|---|---|---|---|
| 8 | 0:00:08.3 | 103 682 | 775 336 |
| 9 | 0:00:27.3 | 439 204 | 3 694 923 |
| 10 | 0:02:28.7 | 1 860 498 | 17 391 050 |
| 11 | 0:17:08.3 | 7 881 196 | 81 036 637 |
| 12 | parallel | 33 385 282 | 374 483 676 |
| 13 | parallel | 141 422 324 | 1 718 533 167 |

### *6.3 CTL properties used*

To test the algorithms developed in the previous sections, we selected six tests that both cover all the algorithms involved and at the same time form a reasonable selection with respect to the possibilities of CTL. The former issue will be discussed below, after we have presented the test cases. However, before we present the test cases, we briefly touch upon the possibilities of CTL.

Recent work on "patterns in CTL" [17] show that liveness conditions (like AG $(\varphi_1 \rightarrow \text{AF } \varphi_2)$), safety constraints (like AG $\varphi$), and reachability properties (like

EF $\varphi$) account for about 43%, 20%, and 7% of all CTL queries, respectively.[7] Hence, we should be able to model check at least such cases efficiently. Notice that the liveness property is nested.

Second, the CTL operators EX, EG, and E($\varphi_1$U$\varphi_2$) form an adequate set of operators for CTL (Katoen J-P, personal communication). Hence if we can model check these cases efficiently, we are in good shape.

The above arguments then lead to the following selection of test cases:

1. **Backward EX test (B-EX).** This test performs a *backward state space generation* using the EX operator implemented with the predecessor function. Given the formula $Sat_{s_0}$, which is only true in the initial marking $s_0$, and the formula $Sat_S$, which is true for every state in the state space $S$, we used the following algorithm:

   1. Sat $(\varphi) \leftarrow \{s_0\}$;
   2. while (Sat $(\varphi) \neq S$)
   3. do Sat $(\varphi) \leftarrow$ EX Sat $(\varphi)$;

   This means that we do a fix-point iteration over $\mathsf{Pred}(\ldots \mathsf{Pred}(\mathsf{Pred}(\{s_0\}))\ldots)$. In the end, this gives us the complete state space if the underlying Markov chain is ergodic. For the dining philosophers problem this test returns all but the two deadlock states in which each philosopher holds exactly the right or the left fork.
2. **Forward EX test (F-EX).** This is the same test as B-EX, however, with the EX operator implemented using the successor function.
3. **Simple EU test (S-EU).** This test again does a backward state space generation by asking the question from which states the initial marking can be reached; in CTL: $\varphi = \mathrm{E}[\mathtt{true}\,\mathrm{U}\,Sat_{s_0}]$. We used the *simple* version of Algorithm 5. Note that this tests the worst-case scenario for an E [. U .] formula, as we start with a set containing only one element and end up with the complete state space as long as the underlying Markov chain is ergodic.
4. **Optimized EU test (O-EU).** This is the same test as S-EU, however, using the optimized variant of the EU algorithm mentioned in Sect. 4.5.
5. **AU test (AU).** This test is specific to the dining-philosophers problem; it computes the set of states for which we can guarantee that at least one philosopher will eat in the future. Phrased in CTL, we have $\varphi = \mathrm{A}[\mathtt{true}\,\mathrm{U}\,\mathit{phil\_eating}]$, where *phil_eating* is an atomic property that marks each state in which at least one philosopher is ready to eat, i.e., owns two forks.
6. **AG test (AG).** This test verifies a liveness property for the dining-philosophers model. We check the CTL formula $\varphi = \mathrm{AG}[\neg \mathit{dead} \rightarrow \mathrm{AF}(\mathit{phil\_eating})]$, where $\mathrm{AG}\psi \equiv \neg\mathrm{E}[\mathtt{true}\,\mathrm{U}\,\neg\psi]$, $\mathrm{AF}\psi \equiv \mathrm{A}[\mathtt{true}\,\mathrm{U}\,\psi]$, and *dead* is an atomic property marking the two deadlock states. Hence this test combines the result from the AU test with an EU test. Obviously this test applies only to the dining-philosophers model.

[7] We use the standard abbreviations: EF $\varphi \equiv$ E (true U $\varphi$), AF $\varphi \equiv$ A (true U $\varphi$), and AG $\varphi \equiv \neg$EF $\neg\varphi$.

**Table 3.** Coverage of the 11 algorithms by the 6 tests

| | **B-EX** | **F-EX** | **S-EU** |
|---|---|---|---|
| Serial | 2, 4 | 1, 3 | 2, 5 |
| Parallel | 7, 8 | – | 7, 9 |
| | **O-EU** | **AU** | **AG** |
| Serial | 2, 5* | $1^+$, 2, 6 | $1^+$, 2, 5, 6 |
| Parallel | 7, 9* | $1^+$, 7, 10, 11 | – |

Table 3 indicates how the selected tests cover the algorithms developed previously. For instance, for the serial **F-EX** test, we used Algorithms 1 and 3. The asterisk in "5*" and "9*" indicates that we used the implementation optimization, as indicated at the end of the explanation of these algorithms. The $1^+$ indicates that we use a slight variant of Algorithm 1 optimized for the use at hand. For instance, in line 5 of Algorithm 5, the successor function is used, however, in a combined fashion with the subsequent test on set inclusion. That is, when a first successor of $s'$ is found that is not included in Sat($\varphi$), no further successors need to be generated and a `false` can be returned. Since **F-EX** was much slower than **B-EX** in the serial case, we decided not to develop a parallel version of it. As becomes clear from this table, all proposed algorithms are covered by our test suite.

### *6.4 Results for the serial algorithms*

#### 6.4.1 Kanban model

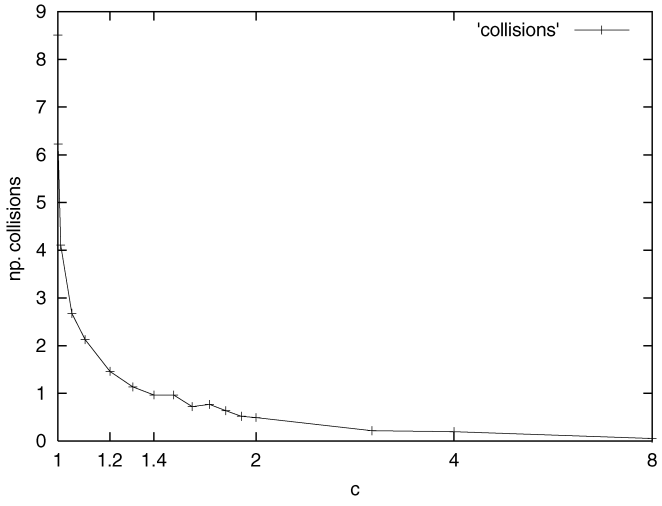
Table 1 shows some statistics for the serial algorithms. In column "Gen. $S$", we list the execution time (wall clock time, in the format hours:min:sec.s) required for the state space generation. We show this time as a reference; it includes the time for writing the state space and the reachability graph to disk. The time required to read the state space from disk and to insert it into the hash table is shown in column "Read $S$". The memory required by our model checker is shown in the next column; it includes the memory for the hash table, internally required bit-vectors, and 13 bits for atomic propositions or subformula evaluated during execution. Column "AP" shows the required time to evaluate four atomic propositions of the form $\#\mathtt{P_i} > 1$. One should note that this is an expensive operation due to the fact that with the PN language CSPL, individual places are addressed using strings, so that string-compares are required to retrieve the number of tokens in a certain marking. The rightmost column

**Table 4.** Serial tests for kanban model

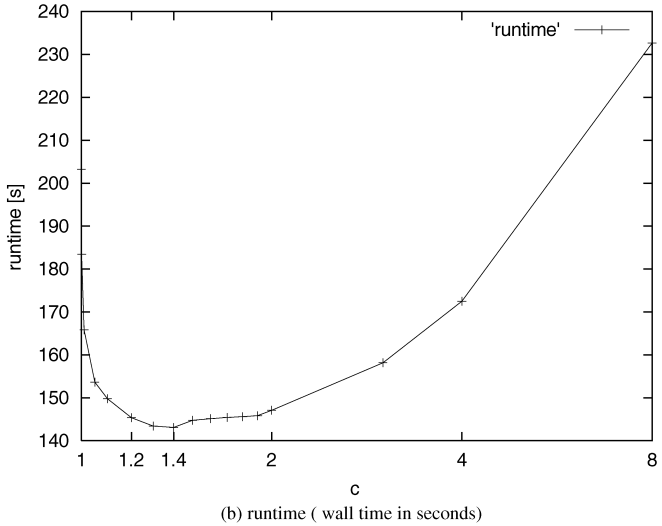
| $N$ | Test | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | B-EX | | F-EX | | S-EU | | O-EU | |
| | Time | Iter. | Time | Iter. | Time | Iter. | Time | Iter. |
| 1 | 0:00.2 | 18 | 0:00.2 | 15 | 0:00.2 | 19 | 0:00.2 | 8 |
| 2 | 0:02.4 | 36 | 0:03.6 | 28 | 0:00.4 | 37 | 0:00.4 | 9 |
| 3 | 0:47.7 | 54 | 1:13.9 | 40 | 0:03.1 | 55 | 0:02.7 | 12 |
| 4 | 9:18.9 | 72 | 14:41.5 | 53 | 0:25.6 | 73 | 0:22.0 | 12 |
| 5 | 1:11:50.0 | 90 | 1:50:22.0 | 65 | 2:35.2 | 91 | 2:09.1 | 16 |
| 6 | 6:22:31.0 | 108 | 10:34:15.0 | 78 | 12:27.1 | 109 | 9:56.9 | 15 |
| 7 | 27:51:34.0 | 126 | 45:25:00.0 | 86 | 48:08.5 | 127 | 37:32.5 | 18 |

shows the time it takes to check a simple logical formula; in this case we construct the set of markings for which $\varphi_1 \wedge \varphi_2$ holds, where $\varphi_1$ and $\varphi_2$ are atomic propositions.

We list a number of test results in Table 4. All model checking tests include the time required to read the state space and insert it into the hash table (as reported in Table 1). We also list the number of iterations required to reach the fix point. Notice the improvement gained by the optimization of directly using newly found states that satisfy $\varphi_1$ in the EU algorithm (**s-EU** vs. **O-EU**).




(a) collision per hash table access




(b) runtime ( wall time in seconds)

**Fig. 5.** Results for S-EU test; kanban model; $N = 5$

Note that for reasonably large models (larger than one million states) the backward state space generation (with knowledge of the state space) via the CTL formula $\varphi = \mathrm{E}[\mathtt{true}\ \mathrm{U}\ \mathrm{Sat}\ (\{\mathtt{s}_0\})]$ is faster than the original state space generation.

As mentioned earlier, all tests were run using hash tables sized 20% larger than the size of the state space ($N_{|S|} = c \cdot |S|$ with $c = 1.2$). We repeated the S-EU test with varying parameter $c$ for the kanban model with $N = 5$ to study its influence on the performance. Figure 5a reports the mean number of collisions per hash table access (which corresponds to a check whether a state exists). For values of $c \geq 1.05$ this number is below 3, and starting with $c \geq 1.2$, it drops below 2. The resulting overall runtime, measured as wall clock time in seconds, is depicted in Fig. 5b. Here we observe that we have the lowest runtime for $1.1 \leq c \leq 2$. For smaller $c$, the runtime increases because of the higher number of collisions. The increasing runtime for larger values of $c$ is due to the required iteration over *all* hash table entries (cf. Algorithm 5). It is important to recognize that the $x$-axis of both figures is scaled logarithmically.

Using a single workstation equipped with 2 GB memory and two 1-GHz Pentium III processors we were also able to generate the state space (in 1:26 h) and to do model checking (the optimized EU test ran for 1:08 h) tests for the case $N = 8$ (almost 134 million states).

### 6.4.2 Dining philosophers

In Table 5 we show some results for the serial algorithms for the dining-philosophers model. Note that both the B-EX and the O-EU tests return all but the two deadlock states (in which all philosophers either have the right fork or the left fork).

Table 6 summarizes the results from the test that involve AU formulas. In the first column we give the number of philosophers $N$. The next four columns present results for the AU test, where we show the number of

**Table 5.** Serial tests for dining-philosophers model

| $N$ | Test | | | |
|---|---|---|---|---|
| | B-EX | | O-EU | |
| | Time | Iter. | Time | Iter. |
| 8 | 0:01:45.1 | 25 | 0:00:07.5 | 8 |
| 9 | 0:10:48.7 | 28 | 0:00:39.4 | 8 |
| 10 | 1:10:54.0 | 31 | 0:03:43.5 | 8 |
| 11 | 8:33:01.6 | 34 | 0:26:06.8 | 9 |

**Table 6.** Serial tests for dining-philosophers model

| $N$ | Test | | | | |
|---|---|---|---|---|---|
| | AU | | | | AG |
| | $\lvert eating \rvert$ | $\lvert \mathrm{Sat}(\varphi) \rvert$ | Time | Iter. | Time |
| 8 | 66 048 | 90 816 | 0:01:02.6 | 6 | 0:01:15.9 |
| 9 | 298 752 | 400 350 | 0:05:29.9 | 7 | 0:06:32.5 |
| 10 | 1 336 324 | 1 743 424 | 0:32:01.3 | 8 | 0:37:25.1 |
| 11 | 5 924 952 | 7 528 950 | 3:52:16.5 | 9 | 4:29:26.3 |

states for which the atomic property *eating* holds and the number of states satisfying the formula checked. Column "Time" and "Iter." show the required wall clock time (in hours:min:sec) and the number of iterations required to reach the fix point. The last column shows the time required for the AG test. A comparison with the column "AU" reveals that the AU test is the most time-consuming part of the AG test.

### *6.5 Distributed algorithms*

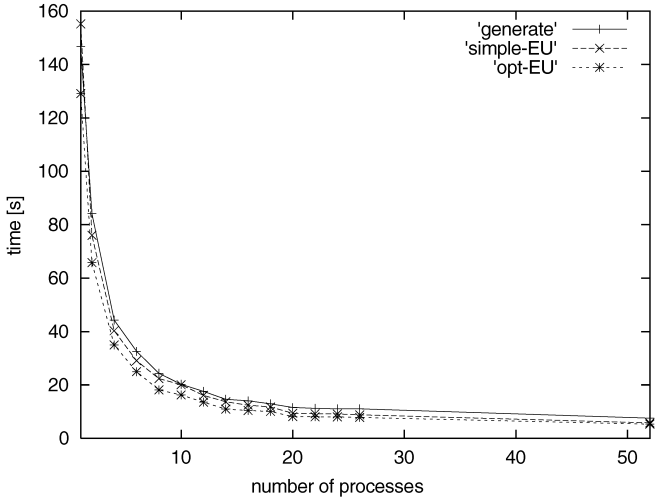
For brevity, we skip the results for the basic logical operators and atomic proposition evaluation as these can trivially be parallelized and the resulting speedups are only dependent on the symmetry of the state space partitioning achieved by the allocation function $A$.

Below, the results given for 2 to 26 processors always use one processor per cluster node, whereas the results for 52 processors use two processors per node. Next to the absolute timings reported, we also report speedups and efficiencies; these are defined as follows. Let $T(1)$ be the runtime of a 1-processor algorithm and $T(n)$, for $n \geq 2$, the runtime of the parallel algorithm using $n$ processors. Then, the speedup with $n$ processors is defined as $S(n) = T(1)/T(n)$. If the algorithm balances the work to be done equally and no additional overhead occurs, the resulting speedup $S(n) = n$. This is called linear speedup. The fraction $E(n) = S(n)/n$ is called the (achieved) efficiency of the parallel algorithm. If one compares the runtime of a parallel program with a specialized serial solution, in contrast to the parallel program running on a single processor only, one speaks of *absolute* speedup and efficiency. We always report absolute measures, as they are known not to overestimate the performance of the parallel algorithms [16].

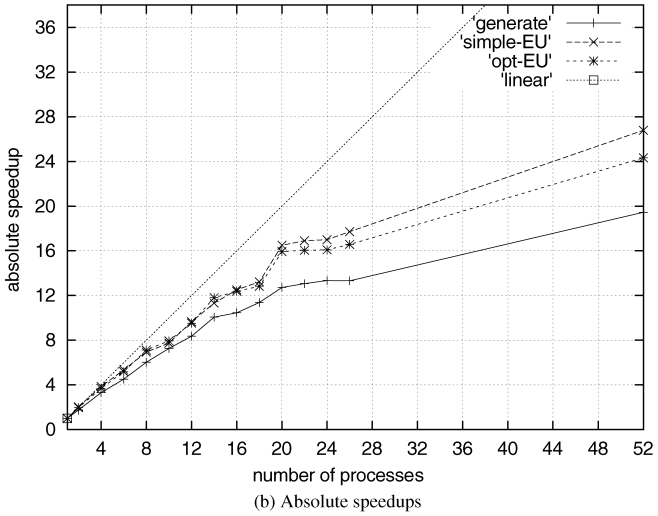
#### 6.5.1 Kanban model

Figure 6a shows the wall clock times for generating the state space and executing the S-EU and O-EU tests for the kanban model with $N = 5$. This is the smallest model for which the measured times are nonnegligible. As one sees in Fig. 6b, even for small models notable speedups can be achieved; we achieve an absolute speedup of around 10 using 12–14 processors for all tests. The speedup for 52 processors is limited by the initialization time, which causes a significant overhead compared to execution times of around 5–7 s. We also ran a variety of "backward EX tests"; the achieved speedups are comparable (and not shown here).

Figures 7 and 8 show the achieved absolute speedups and the absolute efficiencies for the kanban model with $N = 6$ and 7, respectively, which are indeed noticeably better than for $N = 5$. From these figures it can be seen that the obtained degree of parallelism is not as high as for our state space generator. This might be due to the fact that




(a) Wall clock time

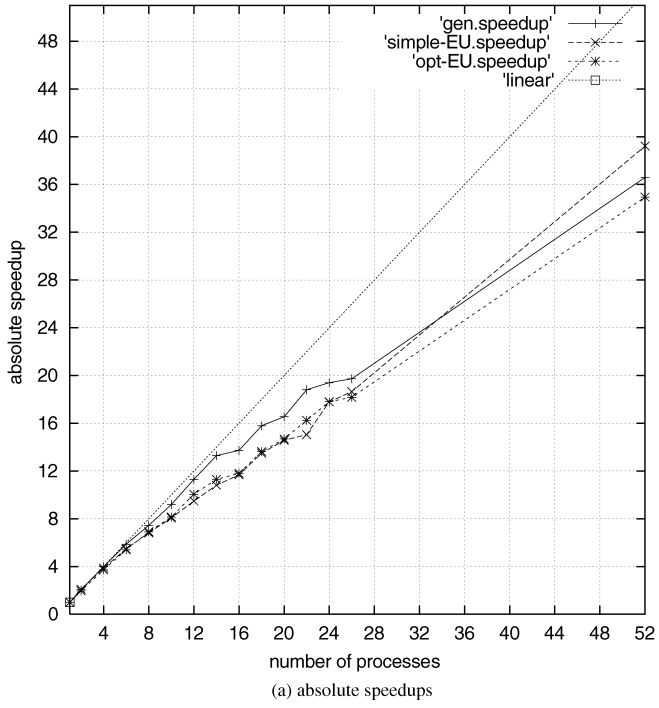



(b) Absolute speedups

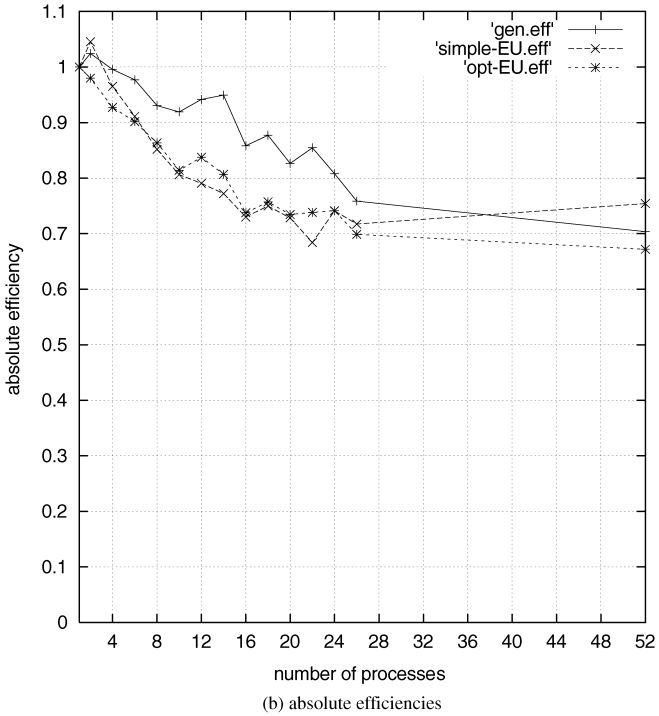
**Fig. 6.** Results for S-EU and O-EU for kanban model with $N = 5$ (2 546 432 states)

we had not yet fully tuned the model checking algorithms, whereas we did so for the state space generation.

For the distributed case the number of iterations to reach a fix point is the same as for the serial case for the EX tests. For the EU tests the number of iterations differs between the processors; it is not even the same for different executions of the program, as it depends on the send/receive timing.

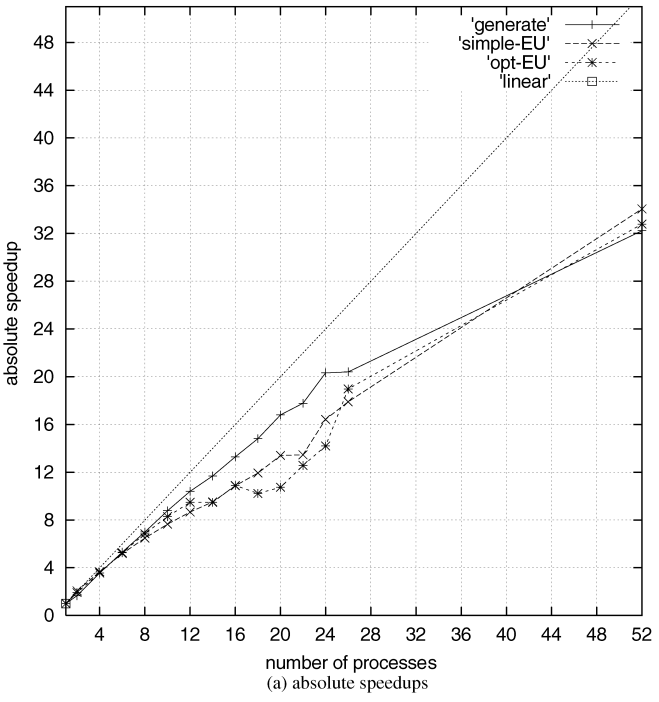



(a) absolute speedups

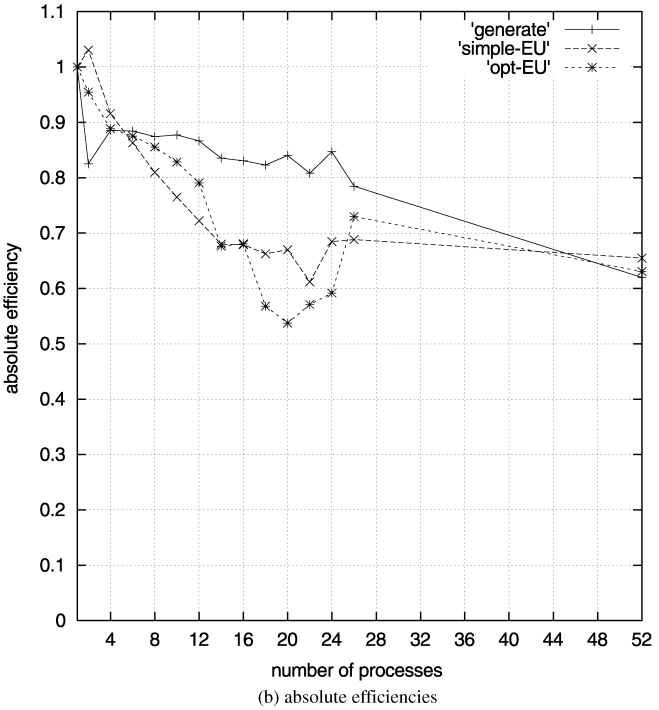



(b) absolute efficiencies

**Fig. 7.** Absolute speedups/efficiencies for S-EU and O-EU for kanban model with $N = 6$ (11 261 376 states)




(a) absolute speedups




(b) absolute efficiencies

**Fig. 8.** Absolute speedups/efficiencies for S-EU and O-EU for kanban model with $N = 7$ (41 644 800 states)

For $N = 8$, one can compare the results from a faster single workstation (Pentium III, 1 GHz, 2 GB RAM) with the result from the cluster using $2 \times 26$ processors (Pentium III, 500 MHz, 512 MB RAM per 2 processors). The O-EU test that required 1:08 h serially completed in 3:51 min using the cluster. This corresponds to a "speedup" (note the slower processors) of 17.6.

As a large test case we ran the O-EU test for the kanban system with $N = 9$ (yielding almost 384 million

states). It took only 15:54 min using all 52 processors. The time required to generate this model was only 15:09 min.

For the Kanban model, we conclude that the distributed algorithms are very efficient, as witnessed by the attained speedups. The induced communication overhead does not severely limit the speedups.

### 6.5.2 Dining philosophers

In Fig. 9 we show the absolute speedup and efficiency for the O-EU test, with 10 and 11 philosophers (two vs. almost eight million states, respectively), again as in the serial case this test returns all but the two deadlock states of the model. We observe that the larger the model, the better the speedup and efficiency.

Furthermore, we also observe superlinear speedups, for which we see two causes. First, when using multiple processors, more processor cache becomes available, so that a better performance can be attained. This is the typical cause for superlinear speedups in parallel computing. Second, specifically for our algorithms, the allocation function used to allocate states to processors induces a different order in which states are seen (or handled) in each processor, when a different number of processors is used. It is well known that for algorithms involving graph searches, the search order has its impact on the algorithm efficiency (see [22] for similar effects in state-space generation). In the cases at hand, we are just lucky that the changed order has a positive effect.

In Fig. 10 we show the absolute speedup and efficiency for the AU and AG tests, again with 10 and 11 philosophers. Obviously, the number of states satisfying these tests is the same as in the serial case (Tables 2 and 6). Notice that the speedups and efficiencies for the AG test are slightly higher than for the AU test, even though the AG test involves a "more complicated" hierarchical CTL expression. The reason for this is that the AG test involves a number of subexpressions (like negations) that can be performed in parallel without communication, hence, of the overall longer computation time, a larger fraction can be parallelized perfectly.

As for the absolute timings, we note the following. The AG test with $N = 11$ takes serially 4:29:26 (notation h:m:s) (Table 6), that is, 16 166 s. With 52 processors and an efficiency of approximately 75%, the parallel solution takes about $16\,166/(52 \times 75\%) = 414.5$ s (the exact measurement value was 416.7 s, or about 6 min and 57 s).

For the dining-philosophers model, we conclude that our algorithms do function very well. Even for the most communication-intensive cases (the AU and AG tests), efficiencies in the range of 60% to 93% with 26 processors and in the range of 52% to 75% with 52 processors can be attained. Furthermore, the larger the models, the higher the obtained efficiencies. Finally, notice that on our cluster, when using 52 processors, performance degradations might occur due to PC-internal bus contention, since two processors have to share a common bus, common main memory, and a common network card (implying half the network bandwidth per processor).

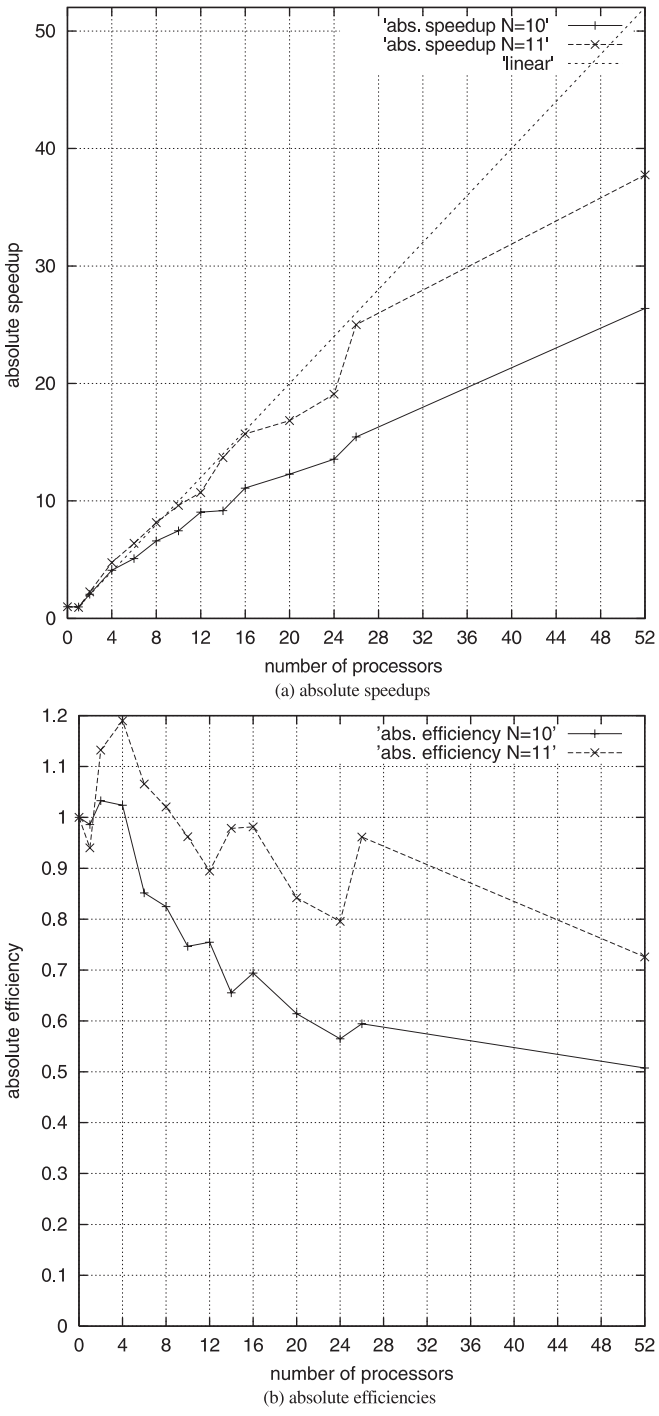
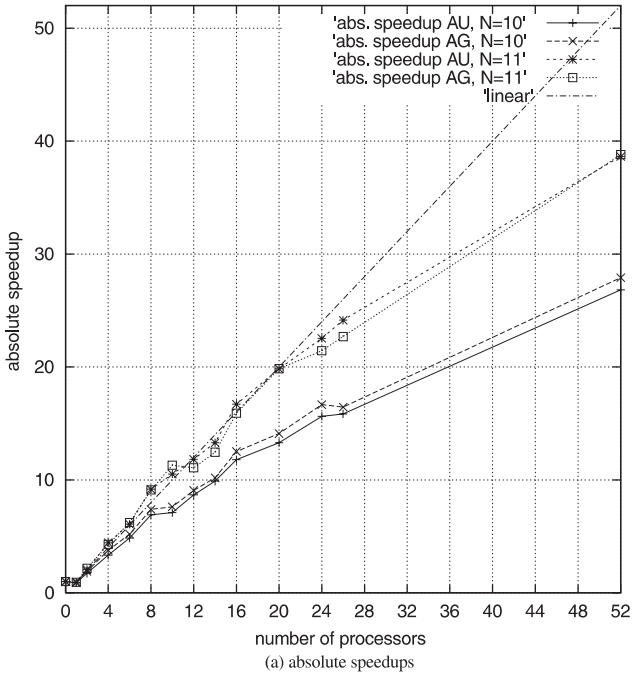


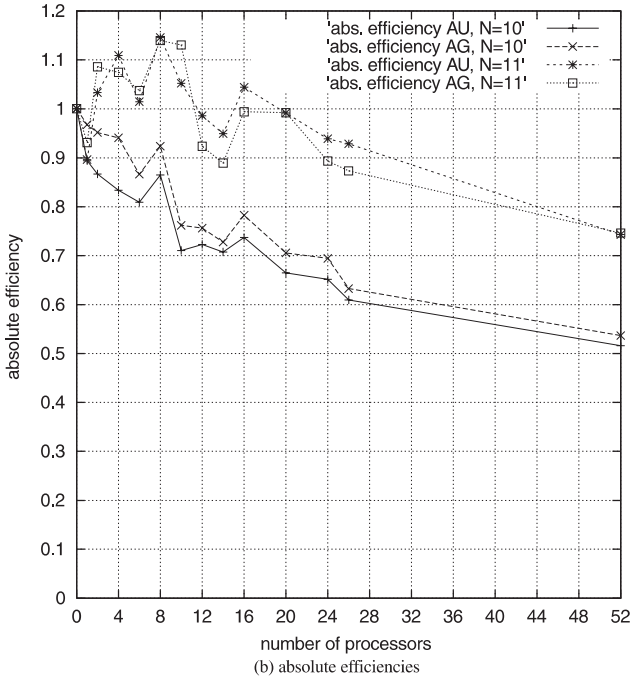

**Fig. 9.** Absolute speedups/efficiencies for O-EU for dining-philosophers model with $N = 10, 11$ (1 860 498 resp. 7 881 196 states)

## 7 Related work

Distributed algorithms for explicit state space *generation* from a high-level system description using multiprocessor




(a) absolute speedups




(b) absolute efficiencies

**Fig. 10.** Absolute speedups/efficiencies for AU for dining-philosophers model with $N = 10, 11$ (1 860 498 resp. 7 881 196 states)

systems or a cluster of workstations have been reported in the last 5 to 10 years, e.g., in [11, 19, 22, 27, 32]. Although successful, the recent achievements in symbolic state space generation techniques have made explicit state space generation techniques less of a research issue.

More recently, new work focusing on parallel and distributed *model checking* has been published. Note that we restrict ourselves to approaches for systems with discrete state spaces, that is, we do not address parallel and distributed approaches to model checking timed automata.

Recently, a number of papers have focused on parallel and distributed model checking for the (alternation-free) $\mu$-calculus. This is especially important since the $\mu$-calculus subsumes the logic CTL; hence, good parallel solutions for the former case imply a good solution for the latter.

Bollig et al. [6, 7] study (local) parallel model checking of the alternation-free $\mu$-calculus on a cluster of workstations using a nonsymbolic approach.[8] Their load distribution approach is, like ours, based on a hashing function assigning states to processors. Although they do not give details about the employed hashing functions, they report good load balance (each processor has to handle at most 10% more or less states than the average). A case study of the alternating-bit protocol is presented (with 1.6 million states), about which they show speedup curves. However, the authors report speedups relative to a 5-processor solution, which should be regarded as rather deceptive [16]. Furthermore, their papers do not present absolute performance measures. Their prototype implementation, written in Haskell, needs about 30 min to generate a state space with a million states on a 52-processor cluster (we do this in less than 1 min on a single machine); clearly, this prototype should just be seen as a "proof of concept".

In [20] the authors propose a distributed and symbolic approach to model checking for the $\mu$-calculus by using a so-called slicing algorithm to achieve good load balance; this paper, however, does not present applications or experimental results. Earlier work of the same authors report good results for parallel reachability analysis [23] as well as for parallel on-the-fly model checking of safety properties [5]. The experimental results reported suggest that the benefit of the parallel approach lies more in the availability of a larger overall main memory than in an improved timing performance (speedup).

Recently, Brim et al. proposed the use of so-called assumptions and techniques known from modular model checking [30] to come up with distributed model checking algorithms for CTL [8]. Although this appears as a very elegant technique with a very nice theoretical foundation, experimental results have not yet been reported.

## 8 Concluding remarks

In this paper we have presented algorithms for model checking CTL over systems specified as Petri nets. We have presented efficient sequential as well as distributed model checking algorithms. The algorithms rely on an explicit representation of the system state space, but the transition relation is recomputed whenever required. This approach allows us to model check very large systems, with hundreds of millions of states, in a fast and efficient

[8] Actually, these authors use the same experimentation platform as we used, that is, the departmental Linux cluster at the RWTH Aachen.

way. The distributed algorithms show efficiencies in the range of 60% to 95%, depending on the test case and case study (size) at hand, even when using 26 or 52 processors. Hence really large systems can be model checked efficiently and in a scalable fashion on a cluster of workstations. A comparison with other sequential algorithms, e.g., with PRISM [29], is planned for the future. We will also consider distributed model checking algorithms for Petri net models including stochastic timing and rewards.

*Acknowledgements.* We thank the anonymous reviewers for their constructive comments and for pointing out some related work.

## References

1. Ajmone Marsan M, Conte G, Balbo G (1984) A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. ACM Trans Comput Sys 2(2):93–122
2. Baier C, Haverkort BR, Katoen J-P, Hermanns H (2000a) Formal specification and verification of performability properties. In: Montanari U, Rolin JDP, Welzl E (eds) Proceedings of ICALP 2000, Geneva, Switzerland, 9–15 July 2000. Lecture notes in computer science, vol 1853. Springer, Berlin Heidelberg New York, pp 780–792
3. Baier C, Haverkort BR, Katoen J-P, Hermanns H (2000b) Model checking continuous-time Markov chains by transient analysis. In: Emerson EA, Sistla AP (eds) Proceedings of CAV 2000, Chicago, 15–19 July 2000. Lecture notes in computer science, vol 1855. Springer, Berlin Heidelberg New York, pp 358–372
4. Bell A (1999) Verteilte Bewertung stochastischer Petrinetze. Diploma thesis, RWTH Aachen, Department of Computer Science, March 1999
5. Ben-David S, Heyman T, Grumberg O, Schuster A (2000) Scalable distributed on-the-fly symbolic model checking. In: Proceedings of the 3rd international conference on formal methods in computer-aided design, Austin, TX, 1–3 November 2000. Lecture notes in computer science, vol 1954. Springer, Berlin Heidelberg New York, pp 390–404
6. Bollig B, Leucker M, Weber M (2001) Parallel model checking for the alternation-free $\mu$-calculus. In: Margaria T, Yi W (eds) Proceedings of the conference on tools and algorithms for the construction and analysis of systems (TACAS 2001), Genoa, Italy 2–6 April 2001. Lecture notes in computer science, vol 2031. Springer, Berlin Heidelberg New York, pp 543–558
7. Bollig B, Leucker M, Weber M (2002) Local parallel model checking for the alternation-free $\mu$-calculus. In: Bosnacki D, Leue S (eds) Proceedings of the 9th international SPIN workshop on model checking of software, Grenoble, France, 11–13 April 2002. Lecture notes in computer science, vol 2318. Springer, Berlin Heidelberg New York, pp 128–147
8. Brim L, Crhova J, Yorav K (2002) Using assumptions to distribute CTL model checking. In: Brim L, Grumberg O (eds) Electronic notes in theoretical computer science, vol. 68. Elsevier, Amsterdam
9. Ciardo G, Tilgner M (1996) On the use of Kronecker operators for the solution of generalized stochastic Petri nets. Technical Report No. 96–35, ICASE, 1996
10. Ciardo G, Muppala JK, Trivedi KS (1989) SPNP: stochastic Petri net package. In: Proceedings of the 3rd international workshop on Petri nets and performance models, Kyoto, Japan, 11–13 December 1989. IEEE Press, New York, pp 142–151
11. Ciardo G, Gluckman J, Nicol D (1998) Distributed state space generation of discrete-state stochastic models. INFORMS J Comput 10(1):82–93
12. Ciardo G, Lüttgen G, Siminiceanu R (2001a) Saturation: an efficient iteration strategy for symbolic state-space generation. Lecture notes in computer science, vol 2031. Springer, Berlin Heidelberg New York, pp 328–342
13. Ciardo G, Lüttgen G, Siminiceanu R (2001b) Saturation: an efficient iteration strategy for symbolic state-space generation. Technical report, ICASE, 2001
14. Clarke EM, Grumberg O, Peled D (1999) Model checking. MIT Press, Cambridge, MA
15. Clarke EM, Emerson EA, Sistla AP (1986) Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans Programm Lang Sys 8(2):244–263
16. Crowl LA (1994) How to measure, present, and compare parallel performance. IEEE Parallel Distrib Technol 2(1):9–25
17. Dwyer MB, Avrunin GS, Corbett JC (1999) Patterns in property specification for finite-state verification. In: Proceedings of the 21st international conference on softwware engineering, Los Angeles, 16–22 May 1999. IEEE Press, New York, pp 411–420
18. Esparza J, Nielsen M (1994) Decidability issues for Petri nets: a survey. Elektron Informationsverarbeit Kybern 30(3):143–160
19. Garavel H, Mateescu R, Smarandache I (2001) Parallel state space construction for model-checking. In: Dwyer M (ed) Proceedings of SPIN 2001, Toronto, 19–20 May 2001. Lecture notes in computer science, vol 2057. Springer, Berlin Heidelberg New York, pp 217–234
20. Grumberg O, Heyman T, Schuster A (2001) Distributed symbolic model checking for $\mu$-calculus. In: Finkel A, Berry G, Comon H (ed) Proceedings of the international conference on computer aided verification, Paris, 18–23 July 2001. Lecture notes in computer science, vol 2102. Springer, Berlin Heidelberg New York, pp 350–362
21. Haverkort BR (1998) Performance of computer-communication systems: a model-based approach. Wiley, New York
22. Haverkort BR, Bell A, Bohnenkamp H (1999) On the efficient sequential and distributed generation of very large Markov chains from stochastic Petri nets. In: Proceedings of the 8th international workshop on Petri nets and performance models, Zaragosa, Spain, 8–10 September 1999. IEEE Press, New York, pp 12–21
23. Heyman T, Geist D, Grumberg O, Schuster A (2000) Achieving scalability in parallel reachability analysis of very large circuits. In: Grumberg O (ed) Proceedings of the 12th international conference on computer-aided verification, Chicago, 15–19 July 2000. Lecture notes in computer science, vol 1855. Springer, Berlin Heidelberg New York, pp 20–35
24. Huth MRA, Ryan MD (2000) Logic in computer science: modelling and reasoning about systems. Cambridge University Press, Cambridge, UK
25. Johr S (2002) MMD-based reachability set generation of stochastic models. Diploma thesis, Department of Computer Science, RWTH Aachen, Aachen, Germany, September 2002
26. Katoen J-P (1999) Concepts, algorithms, and tools for model checking, IMMD Berichte, vol 32. Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany, June 1999
27. Knottenbelt W, Mestern M, Harrison P, Kritzinger P (1998) Probability, parallelism and the state space exploration problem. In: Puigjaner R, Savino NN, Serra B (eds) Computer performance evaluation. Lecture notes in computer science, vol 1469. Springer, Berlin Heidelberg New York, pp 165–179
28. Knuth DE (1973) The art of computer programming, vol 3: Sorting and searching. Addison-Wesley, Reading, MA
29. Kwiatkowska M, Norman G, Parker D (2002) PRISM: probabilistic symbolic model checker. In: Proceedings of TOOLS 2002, London, 14–17 April 2002. Lecture notes in computer science, vol 2324. Springer, Berlin Heidelberg New York, pp 200–204
30. Laster K, Grumberg O (1998) Modular model checking of software. In: Proceedings of the conference on tools and algorithms for the construction and analysis of systems (TACAS 1998), Lisbon, Portugal, 28 March–4 April 1998. Lecture notes in computer science, vol 1384. Springer, Berlin Heidelberg New York, pp 20–35
31. MPICH – a portable implementation of MPI. `www.mcs.anl.gov/mpi/mpich`
32. Stern U, Dill DL (1997) Parallelizing the Mur$\varphi$ verifier. In: Grumberg O (ed) Proceedings of the conference on computer aided verification, Haifa, Israel, 22–25 June 1997. Lecture notes in computer science, vol 1254. Springer, Berlin Heidelberg New York, pp 256–267