

# Local Quantitative LTL Model Checking<sup>\*</sup>

J. Barnat, L. Brim, I. Černá, M. Češka, and J. Tůmová

Faculty of Informatics, Masaryk University  
Brno, Czech Republic  
{barnat,brim,cerna,xceska,xtumova}@fi.muni.cz

**Abstract.** Quantitative analysis of probabilistic systems has been studied mainly from the global model checking point of view. In the global model-checking, the goal of verification is to decide the probability of satisfaction of a given property for all reachable states in the state space of the system under investigation. On the other hand, in local model checking approach the probability of satisfaction is computed only for the set of initial states. In theory, it is possible to solve the local model checking problem using the global model checking approach. However, the global model checking procedure can be significantly outperformed by a dedicated local model checking one. In this paper we present several particular local model checking techniques that if applied to global model checking procedure reduce the runtime needed from days to minutes.

## 1 Introduction

System design techniques employing probability are becoming widely used. They provide designers with reasonably efficient means to break symmetry in the system or to implement randomized algorithms. Probabilistic actions are also used for modeling various nondeterministic aspects such as human unpredictable decisions, occurrence of external stimuli, or simply the presence of hardware errors. As the interest in the probabilistic systems is growing, supported mainly by their potential practical use, there is also increased interest in formal techniques for their analysis and verification, model checking in particular.

There are two different tasks related to model checking over probabilistic systems. Given a formula and probabilistic system, the so called *qualitative* analysis refers to the problem of deciding whether the probabilistic system satisfies the formula with the probability one. On the other hand, the so called *quantitative* model checking refers to the problem of deciding the maximal and minimal probability the given formula is satisfied for the probabilistic system. For model checking linear time properties, the qualitative problem can be solved similarly to the nondeterministic case, i.e. using automata-based approach. The problem reduces to the problem of the detection of an *Accepting End Component* (AEC) in the graph of the underlying product of the probabilistic system and the  $\omega$ -regular automaton expressing the (negation of) the verified property [22, 11]. For

---

<sup>\*</sup> This work has been partially supported by the Grant Agency of Czech Republic grant No. 201/06/1338 and the Academy of Sciences grant No. 1ET408050503.

the quantitative case the model checking procedure is a little bit more complex [3, 12]. Similarly to the qualitative case, the probabilistic system is multiplied with the semi-deterministic  $\omega$ -regular property automaton and all the AECs are identified in its underlying graph. After that, the graph is transformed into a linear programming problem (set of inequalities over states of the probabilistic system and an objective function to be maximized). Every variable in the linear programming instance corresponds to a state in the system in the sense that the value computed for the variable is exactly the maximal probability of satisfaction of the examined property, if the property is evaluated from the particular state. States in an AEC satisfy the examined property with the probability one.

Both qualitative and quantitative analysis of probabilistic systems has been studied mainly from the *global model checking* point of view. In the global model-checking, the goal of verification is to decide the probability of satisfaction of a given property for all reachable states in the state space of the system under investigation. On the other hand, in *local model checking* approach the probability of satisfaction is computed only for the set of initial states. In theory, it is possible to solve the local model checking problem using the global model checking approach. However, the global model checking procedure can be significantly outperformed by a dedicated local model checking one. It is a well-known fact that from practical point of view, the system designers are often interested in the probability of satisfaction of the property for some particular states only (initial state most typically). This is not taken into account in the general global model checking scheme as suggested in [3, 12].

There are several software tools performing qualitative and/or quantitative probabilistic model checking. Probably the most established probabilistic model checker is the *symbolic* model checker PRISM [16]. It provides support for automated analysis of a wide range of quantitative properties for three types of probabilistic models: discrete-time Markov chains, continuous-time Markov chains and Markov decision processes (MDPs). The property specification language of PRISM incorporates the temporal logics PCTL [15] and CSL [1] as well as extensions for quantitative specifications and costs/rewards. As for *enumerative* approach to model checking, the model checker to be mentioned is LIQUOR [10]. LIQUOR is capable of verifying probabilistic systems modeled as ProbMeLa programs. ProbMeLa is a probabilistic guarded command language with an operational semantics based on finite MDPs. LIQUOR allows qualitative and/or quantitative analysis for  $\omega$ -regular linear time properties. The tool follows the standard automata-based model checking approach and involves partial order reduction technique for MDPs [4] to fight the state explosion problem. Recently, a parallel enumerative probabilistic model checker – PROBDIVINE, has been released [5]. Likewise LIQUOR, PROBDIVINE provides means for verification of quantitative and qualitative linear time properties of MDPs. The unique feature of PROBDIVINE is its capability of employing combined power of multiple CPU cores available on latest hardware systems to solve large verification problems. All the techniques presented in this paper have been implemented and experimentally evaluated using the PROBDIVINE model checker. Yet another tool for

verification of MDPs is the model-checker RAPTURE [8]. It employs an automatic abstraction refinement and essential state reduction techniques to fight the state explosion problem [8].

As the main contribution of this paper we introduce several techniques that allow to improve the general *quantitative* verification procedure using the locality of the model checking goal. These local model checking techniques can be applied to the global model checking procedure resulting in a significant speed-up as indicated by our experimental evaluation. In addition, the locality of the techniques supports their natural integration into a parallel tool, giving thus further advantages in terms of speed and scalability.

The rest of the paper is organized as follows. Section 2 states the necessary definitions and recalls the general scheme of quantitative model checking procedure. Section 3 introduces our new techniques to improve the general verification scheme, Section 4 reports on experimental evaluation of these techniques, and Section 5 concludes the paper.

## 2 Preliminaries

In this subsection, we briefly review fundamentals of LTL model checking over finite state probabilistic systems and fix some notation.

### 2.1 Probabilistic model checking

*Markov decision processes* (MDPs) are used as the standard modeling formalism for asynchronous probabilistic systems, supporting both nondeterminism and probability. A Markov decision process [13, 21, 22], is a tuple  $M = (S, Act, P, init, AP, L)$ , where  $S$  is a finite set of states,  $Act$  is a finite set of actions,  $P : (S \times Act \times S) \rightarrow [0, 1]$  is a probability matrix,  $init \in S$  is the initial state,  $AP$  is a finite set of atomic propositions, and  $L : S \rightarrow 2^{AP}$  is a labeling function.  $Act(s)$  denotes the set of actions that are enabled in the state  $s$ , i.e. the set of actions  $\alpha \in Act$  such that  $P(s, \alpha, t) > 0$  for some state  $t \in S$ . For any state  $s \in S$ , we require that  $Act(s) \neq \emptyset$  and  $\forall \alpha \in Act(s). \sum_{s' \in S} P(s, \alpha, s') = 1$ .

An infinite run of an MDP is a sequence  $\tau = s_0, \alpha_1, s_1, \alpha_2, \dots \in (S \times Act)^\omega$  such that  $\alpha_i \in Act(s_{i-1})$ . A trajectory of  $\tau$  is the word  $L(s_0), L(s_1), L(s_2), \dots$  over the alphabet  $2^{AP}$  obtained by the projection of  $\tau$  to the state labels.

The intuitive operational semantics of an MDP is as follows. If  $s$  is the current state then an action  $\alpha \in Act(s)$  is chosen nondeterministically and is executed leading to a state  $t$  with probability  $P(s, \alpha, t)$ . We refer to  $t$  as an  $\alpha$ -*successor* of  $s$  if  $P(s, \alpha, t) > 0$ . State  $s$  is called *deterministic* if exactly one action is enabled in  $s$ . If all states of an MDP are deterministic, the MDP is called *Markov chain*. To resolve the nondeterminism of an MDP a *scheduler* function is used. We consider deterministic history dependent schedulers which are given by a function  $D$  assigning an action  $D(\sigma) \in Act(s_n)$  to every finite run  $\sigma = s_0, \alpha_1, \dots, \alpha_n, s_n$ . Given a scheduler  $D$ , the behavior of  $M$  under  $D$  can be formalized as a Markov chain.

Let  $M$  be a Markov Chain,  $s \in S$  be a state of  $M$ , and  $X$  be a set of runs of  $M$  originating at  $s$ . We define *the probability of the set  $X$*  as a measure of the set  $X$  in the set of all runs of  $M$  originating at  $s$ . A set  $X$  of runs of a Markov Chain  $M$  is called *basic cylinder set* if there is a prefix  $s_0, \alpha_1, \dots, \alpha_n, s_n$  such that  $X$  contains exactly all runs of  $M$  with that prefix. The probability measure of a basic cylinder set with prefix  $s_0, \alpha_1, \dots, \alpha_n, s_n$  is then

$$\prod_{i=0}^{n-1} P(s_i, \alpha_{i+1}, s_{i+1}).$$

If the set  $X$  of runs of  $M$  is not a basic cylinder set, its measure is determined as a sum of measures of maximal (w.r.t. inclusion) basic cylinder sets fully contained in  $X$  [11].

In this paper we focus on the *quantitative model checking* of MDPs against properties specified in Linear temporal logic (LTL). Formulas of LTL are built over a set  $AP$  of atomic propositions and are closed under the application of Boolean connectives, the unary temporal connective  $X$  (next), and the binary temporal connective  $U$  (until). LTL is interpreted over computations. A *computation* is a function  $\pi : \omega \rightarrow 2^{AP}$ , which assigns truth values to the elements of  $AP$  at each time instant and as such it can be viewed as an infinite word over the alphabet  $2^{AP}$ . For an LTL formula  $\varphi$ , we denote by  $\mathcal{L}(\varphi)$  the set of all computations satisfying  $\varphi$ .

A run of a Markov chain satisfies the formula  $\varphi$ , if the trajectory of the run is in  $\mathcal{L}(\varphi)$ . A Markov Chain  $M$  satisfies the formula  $\varphi$  with probability  $p$ , if the set of runs of  $M$  satisfying the formula has the probability  $p$ . An MDP  $M$  satisfies the formula  $\varphi$  with the probability at least  $p$  (at most  $p$ ) if for every scheduler  $D$ ,  $M$  under  $D$  satisfies the formula with the probability at least  $p$  (at most  $p$ ). The problem of quantitative model checking is to determine the minimal and/or maximal probability that an MDP satisfies a given property. Note that for the computation of the minimal and/or maximal probability that an MDP satisfies an  $\omega$ -regular property, it is sufficient to consider only history independent schedulers [12].

The goal of the *global* quantitative model checking is to calculate the minimal and/or maximal probability of the satisfaction of the property for every state  $s$  of an MDP. The goal of the *local* quantitative model checking is, however, to determine the minimal and/or maximal probability of satisfaction of the property for the initial state only.

A Büchi automaton is a tuple  $A = (\Sigma, Q, q_{init}, \delta, F)$ , where  $\Sigma$  is a finite alphabet,  $Q$  is a finite set of states,  $q_{init} \in Q$  is an initial state,  $\delta \subseteq Q \times \Sigma \times Q$  is a transition relation, and  $F \subseteq Q$  is a set of *accepting states*. A run of  $A$  over an infinite word  $w = a_1 a_2 \dots \in \Sigma^\omega$  is a sequence  $q_0, q_1, \dots$ , where  $q_0 = q_{init}$  and  $(q_{i-1}, a_i, q_i) \in \delta$  for all  $i \geq 1$ . Let  $\text{inf}(\rho)$  denote the set of states that appear in the run  $\rho$  infinitely often. A run  $\rho$  is accepting iff  $\text{inf}(\rho) \cap F \neq \emptyset$ . A state  $s \in Q$  of a Büchi automaton  $A$  is called *deterministic* if and only if for all  $a \in \Sigma$  there is at most one  $s' \in A$  such that  $(s, a, s') \in \delta$ . A Büchi automaton is *deterministic*

in the limit if and only if all the accepting states and their descendants are deterministic [11].

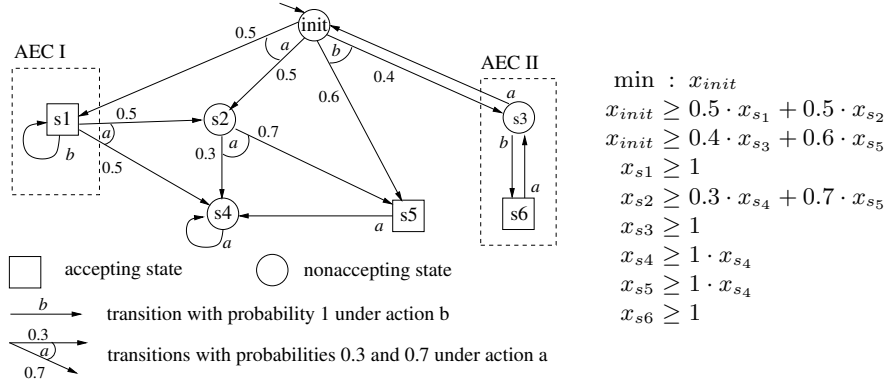
We use the automata based approach to probabilistic LTL model checking. Given an LTL formula  $\varphi$ , it is possible to build a Büchi automaton  $A$  with  $2^{O(|\varphi|)}$  states such that  $L(A) = \mathcal{L}(\varphi)$  [23]. Moreover, for any Büchi automaton  $A$  with  $n$  states a Büchi automaton  $B$  with  $2^{O(n)}$  state such that  $B$  is *deterministic in the limit* and  $L(A) = L(B)$  can be built [11]. Similarly to model checking non-probabilistic systems, the model is synchronized with the automaton corresponding to the negation of the formula in the case we are interested in the minimal probability or with the automaton corresponding to the formula in the case we are interested in the maximal probability. However, unlike the non-probabilistic case, automata which are deterministic in the limit have to be used instead of non-deterministic Büchi automata.

Let  $M = (S, Act, P, s_0, AP, L)$  be an MDP and let  $A = (Q, 2^{AP}, q_0, \Delta, F)$  be a Büchi automaton. The synchronized product of  $M$  and  $A$  is an extended MDP  $M \times A = (S \times Q, Act_{M \times A}, P_{M \times A}, init, AP, L_{M \times A}, Acc)$ , where  $Act_{M \times A}((s, p)) = Act(s)$ ,  $P_{M \times A}((s, p), \alpha, (t, q)) = P(s, \alpha, t)$  if  $(p, L(s), q) \in \delta$  or 0 otherwise,  $init = (s_0, q_0)$ ,  $L_{M \times A}((s, t)) = L(s)$ , and  $Acc = S \times F$  is the set of accepting states. Note that the synchronized product is not a regular MDP as it distinguishes between accepting and non-accepting states and may contain states without enabled actions.

In order to describe the algorithmic solution to the quantitative LTL model checking we often view an MDP or MDP synchronized with a Büchi automaton as a graph. Therefore, we recall some basic notions from the graph theory. A state  $s'$  is *reachable* from a state  $s$  in a set of states  $R \subseteq S$ , denoted as  $s \rightsquigarrow_R^+ s'$  iff there is a sequence of states  $s_0, s_1, \dots, s_k \in R$  such that  $s = s_0, s' = s_k$  and for all  $0 \leq i < k$  there is an action  $\alpha \in Act(s_i)$  such that  $P(s_i, \alpha, s_{i+1}) > 0$ . A set of states  $R$  is *strongly connected* if for all  $r, r' \in R : r \rightsquigarrow_R^+ r'$  or  $|R| = 1$ . A *strongly connected component* (SCC) is a maximal strongly connected set of states. The graph of strongly connected components of  $G$  is called the *quotient graph* of  $G$ . An SCC  $C$  is *trivial* if  $|C| = 1$ . An SCC is *terminal* if it has no successors in the quotient graph. For every component  $C$  let  $Input(C) = \{c \in C \mid \text{there is an SCC } C' : \exists c' \in C' : \exists \alpha \in Act(c') : P(c', \alpha, c) > 0\}$  if  $init \notin C$  otherwise  $Input(C) = \{init\}$ . Furthermore, for each nonterminal component  $C$  we define  $Output(C) = \{s \in S \setminus C \mid \exists c \in C : \exists \alpha \in Act(c) : P(c, \alpha, s) > 0\}$ .

Given an MDP graph  $G$ , an *accepting end component* (AEC) is a maximal set  $C$  of states of  $G$  that forms (not necessary maximal) strongly connected component in  $G$  such that  $C \cap Acc \neq \emptyset$  and if there is an enabled action  $\alpha$  in a state of the component, the component contains either all the  $\alpha$ -successors or none of them [12]. From [13, 14] it follows that for any state  $s$  of an MDP graph of  $M \times A$  that belongs to an AEC, there exists a scheduler  $D$  such that the probability measure of runs originating in  $s$  and remaining in the AEC is 1 in  $M$  under  $D$ .

Let  $s$  be a state in the MDP product graph. We define the maximal probability  $x_s$  of reaching an AEC from  $s$  as follows. If  $s$  belongs to an AEC,  $x_s = 1$ ,



**Fig. 1.** MDP and its corresponding local linear programming problem

if no AEC is reachable from  $s$ ,  $x_s = 0$ . For remaining cases the value of  $x_s$  can be calculated by solving the linear programming problem with inequalities

$$x_s \geq \sum_{v \in S \times Q} P_{M \times A}(s, \alpha, v) \cdot x_v \quad \forall \alpha \in Act_{M \times A}(s)$$

minimizing the objective function  $f = \sum_{u \in S \times Q} x_u$ . For more details see [3, 12].

Note that in the context of local model checking the objective function can be simplified to  $f = x_{init}$ . An example is given in Figure 1.

After the solution of the linear programming problem is found,  $x_{init}$  contains the value of maximal probability an AEC is reached from the state  $init$ . If the MDP was synchronized with the automaton corresponding to the negation of a formula  $\varphi$ , the minimal probability the MDP satisfies the formula  $\varphi$  is  $1 - x_{init}$ . If the MDP was synchronized with the automaton corresponding to a formula  $\psi$  (without negation), the maximal probability the MDP satisfies  $\psi$  is  $x_{init}$ .

## 2.2 Algorithm

The algorithm for finding all AECs was introduced in [11, 12] and it was based on recursive decomposition of MDP graph into SCCs. Our approach employs a parallel adaptation of the algorithm of Bianco and de Alfaro (BdA) [7] that computes a set of states for which there exists a scheduler such that the maximal probability of reaching an AEC from the set is equal to 1. Clearly, this set can be used instead of the set of all AECs. Henceforward, the set is referred to as  $AS$ .

The algorithm maintains an *approximation set* of states that may belong to an AEC. The algorithm repeatedly refines the approximation set by locating and removing states that cannot belong to an AEC, we call this a *pruning step*. The algorithm for quantitative verification is obtained by a modification of BdA. As the final approximation set is  $AS$ , the linear programming problem is extended with inequalities  $x_u \geq 1$  for all  $x_u \in AS$ . The overall scheme of how the algorithm proceeds is given as Algorithm 1.

---

**Algorithm 1** Scheme of algorithm for local quantitative analysis

---

- 1: compute the set  $AS$  using BdA algorithm
  - 2: create the linear programming problem  $LP$
  - 3: compute the solution of  $LP$
  - 4: **return**  $x_{init}$
- 

### 3 Local Model Checking Techniques

In this section we introduce three optimization techniques that can significantly speed-up the verification process. We also propose a way how these techniques can be employed in a parallel environment, shared-memory multi-core architectures in our case. This is in particular very important in handling very large real-life systems in practice.

#### 3.1 Minimal Subgraph Identification

The first of the proposed algorithmic modifications helps to reduce the size of the linear programming problem by pruning the MDP product  $M \times A$  into the so called minimal subgraph.

The probability of a state depends on the probabilities of its successors. However, once we know that the probability of a state is 1, we do not need to know the exact probabilities of its successors. Also, the probability of a state is 0 if no state with probability 1 can be reached from it. Henceforward, we say that a state is *relevant* if it is on a path from an initial state to a state with probability 1 such that the path does not contain any other state with probability 1. The last state on the path is referred to as a *seed*. Relevant states define in a natural way a slice in the original MDP (see the example in Figure 2). We call this slice a *minimal subgraph*. The probability of the initial state is fully determined by the states in the minimal subgraph only.

With the minimal subgraph we associate the linear programming problem  $mLP$  to be minimized in the following way. Let  $init, s_0, s_1, \dots, s_{r-1}, s_r$  be a path in the minimal subgraph from the initial state  $init$  to a seed  $s_r$ . We add to  $mLP$  the inequalities of form:

$$\begin{aligned}x_{init} &\geq \dots + p_{s_0}x_{s_0} + \dots \\x_{s_0} &\geq \dots + p_{s_1}x_{s_1} + \dots \\&\vdots \\x_{s_{r-1}} &\geq \dots + p_{s_r}x_{s_r} + \dots\end{aligned}$$

It is not difficult to prove that pruning the original MDP graph into the minimal one does not have any influence on the solution of the linear programming problem.

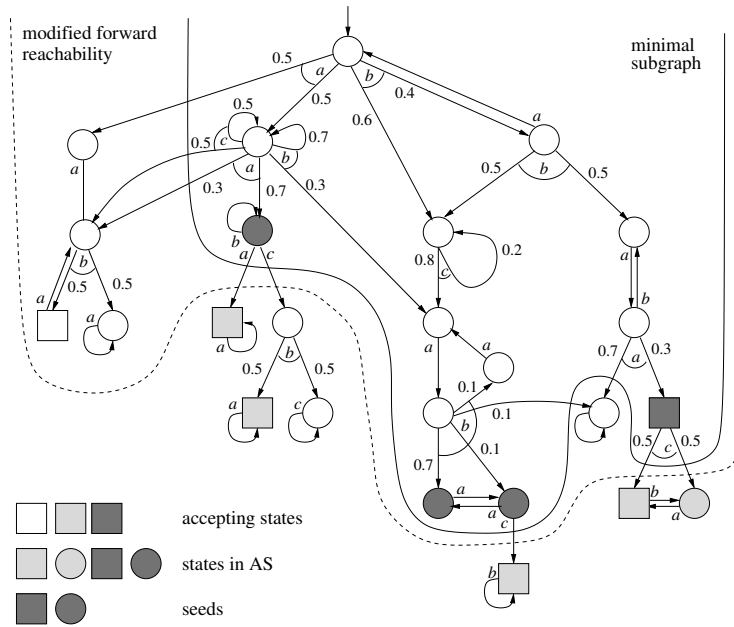


Fig. 2. Minimal subgraph identification.

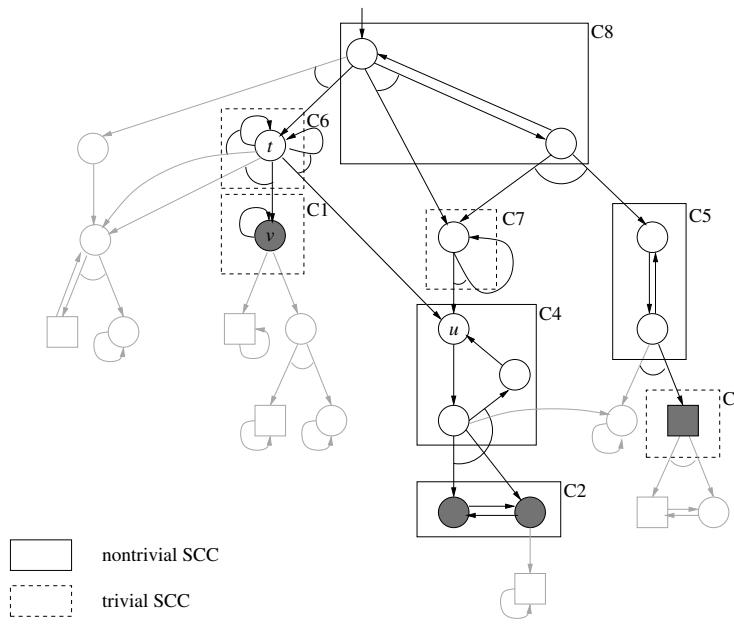


Fig. 3. SCC decomposition.

**Lemma 1.** *Let  $M$  be a synchronous product of MDP and Büchi automaton, and  $MG$  its minimal subgraph. The solution of linear programming problem  $LP$  is equal to the solution of linear programming problem  $mLP$ .*

Having computed  $AS$  we can identify relevant states, i.e. the minimal subgraph, as follows. First, we run a forward reachability from the initial state that does not explore states beyond a state from  $AS$ . States from  $AS$  visited in this reachability are the seeds. Second, we run backward reachability from seeds to identify the minimal subgraph. All states visited by the backward reachability are relevant states. In this manner we omit states whose probability of reaching an AEC equals to zero. The result of applying both forward and backward reachability to obtain the minimal subgraph is illustrated in Figure 2.

### 3.2 Iterative Computation

Another practical technique is to decompose the given problem into simpler subtasks. In this way we have a good chance to end-up with a set of smaller linear programming problems that can be solved much faster.

The core idea is to decompose the minimal subgraph into SCCs, create the appropriate quotient graph, and then iteratively solve the linear programming problem by solving the subproblems given by the individual components in a bottom-up manner.

Some subtasks can be solved independently, which provides a basis for an effective parallel procedure as described in Subsection 3.4. Furthermore, we show in Subsection 3.3 that some subtasks can be solved without employing an external LP solver. Iterative computation can lead to a significant speed-up as compared to the computation of the entire LP problem (see Section 4).

Let us consider a minimal subgraph  $MG$ , its strongly connected components component  $C$  is formed by all inequalities  $x_s \geq \dots$  from  $mLP$  such that  $s \in C$ . The objective function of  $LP_C$  minimizes the sum of variables  $x_s$ ,  $s \in Input(C)$ , i.e. states with a predecessor outside the component, or the value  $x_{init}$  in case  $init \in C$ .

The solution of  $LP_C$  for each component  $C$  depends only on  $C$  itself and on the states in  $Input(C_t)$  for each immediate successor component  $C_t$  of  $C$ , as only variables corresponding to these states appear in the inequalities. This means, that for a terminal SCC  $T$  we can find the solution of  $LP_T$  directly. Once we have solutions for all successor components  $C_t$  of  $C$ , we can substitute all the variables  $x_s$ , such that  $s \notin C$ , with already computed values.  $LP_C$  does not depend on components  $C_t$  any more and the solution of  $LP_C$  can thus be computed. We call the SCC  $C$  *solved* if  $LP_C$  has been solved, i.e. the values  $x_s$  for all  $s \in Input(C)$  have been computed. An unsolved SCC  $C$  is called *prepared* if for all  $t \in Output(C)$  the state  $t$  is in a solved SCC.

**Lemma 2.** *For each  $s \in Input(C)$ , the solution of  $LP_C$  assigns to  $x_s$  the value equal to the maximal probability that the set  $AS$  is reachable from  $s$ .*

**Corollary 1.** *For the component  $C$  containing the initial state  $init$ , the solution of  $LP_C$  assigns to  $x_{init}$  the value equal to the maximal probability AEC is reachable from  $init$ .*

The pseudo-code of the iterative computation is described in Algorithm 2.

---

**Algorithm 2** Iterative computation

---

**Require:** minimal subgraph  $MG$

```

1: decompose  $MG$  into SCCs
2: build the quotient graph of  $MG$ 
3: while there is an unsolved SCC do
4:   compute the set  $P$  of prepared SCCs
5:   for all  $C \in P$  do
6:     create the linear programming problem  $LP_C$ 
7:     substitute for  $x_s$  such that  $s \in Output(C)$  in  $LP_C$ 
8:     compute the solution of  $LP_C$ 
9:     mark  $C$  as solved
10:  end for
11: end while
12: return  $x_{init}$ 

```

---

Figure 3 shows the decomposition into strongly connected components. Components  $C1, C2$  and  $C3$  are terminal and thus prepared. After they are solved, components  $C4$  and  $C5$  become prepared. In the next iteration of the algorithm, components  $C6$  and  $C7$  are prepared and finally, after their solution, the component  $C8$  containing the initial state is ready to be solved.

### 3.3 Trivial SCC

Let us suppose we perform the iterative computation on the minimal subgraph  $MG$  as introduced in the previous Subsection, and let the next *prepared* SCC to be solved is a *trivial* strongly connected component  $T = \{t\}$  with the corresponding linear programming subtask  $LP_T$ . In the following we show, that the linear programming subtask  $LP_T$  can be solved without employing an external LP solver.

Let us firstly recall that due to deAlfaro [12] there is a *history independent* scheduler that yields the maximum value for the state  $t$ . We denote by  $x_t^\alpha$  the probability of the state  $t$  under the history independent scheduler choosing the action  $\alpha \in Act(t)$  whenever the state  $t$  is visited. Since it is sufficient to consider only history independent schedulers for the computation of the probability  $x_t$  of the state  $t$ , it follows directly that

$$x_t = \max_{\alpha \in Act(t)} x_t^\alpha.$$

Lemma 3 says how to compute the value of  $x_t^\alpha$ . Before we state the lemma, we introduce the necessary notation. Suppose an action  $\alpha$  to be executed. Let

$u_0, u_1, \dots, u_n$  be the  $\alpha$ -successors of  $t$  that are outside the component  $T$ . Each  $u_i$  is reached with the probability  $P(t, \alpha, u_i)$  for  $0 \leq i \leq n$ . Let us denote these probabilities  $p_{u_0}^\alpha, p_{u_1}^\alpha, \dots, p_{u_n}^\alpha$ , respectively. Since the states are outside the component  $T$ , the probability values for these states are already known and are referred to as  $v_{u_0}, v_{u_1}, \dots, v_{u_n}$ . Furthermore, we denote the probability  $P(t, \alpha, t)$  by  $p_t^\alpha$ .

**Lemma 3.** *Let  $pv_u^\alpha = p_{u_0}^\alpha v_{u_0} + p_{u_1}^\alpha v_{u_1} + \dots + p_{u_n}^\alpha v_{u_n}$ . Then,*

$$x_t^\alpha = \begin{cases} \frac{pv_u^\alpha}{1-p_t^\alpha} & \text{if } p_t^\alpha \neq 1 \\ 0 & \text{otherwise.} \end{cases}$$

*Proof.* For a Markov chain the following holds:

$$\sum_{i=0}^n p_{u_i}^\alpha + p_t^\alpha \leq 1$$

Therefore, if  $p_t^\alpha = 1$  then  $p_{u_i}^\alpha = 0$  for all  $0 \leq i \leq n$  and thus  $x_t^\alpha = 0$ . Otherwise

$$\begin{aligned} x_t^\alpha &= pv_u^\alpha + p_t^\alpha (pv_u^\alpha) + (p_t^\alpha)^2 (pv_u^\alpha) + (p_t^\alpha)^3 (pv_u^\alpha) + (p_t^\alpha)^4 (pv_u^\alpha) + \dots = \\ &pv_u^\alpha (1 + p_t^\alpha + (p_t^\alpha)^2 + (p_t^\alpha)^3 + (p_t^\alpha)^4 + \dots) = pv_u^\alpha \frac{1}{1-p_t^\alpha} \quad \square \end{aligned}$$

To compute the value of  $x_t$  we enumerate the values  $x_t^\alpha$  according to the previous Lemma and compute their maximum. For an example, we refer to Figures 2 and 3. The component  $C6$  containing the state  $t$  is a trivial one. After the components  $C1$  and  $C4$  are solved, we have  $v_u = 0.9$ ,  $u \in Input(C4)$  and  $v_v = 1$ ,  $v \in Input(C1)$ . The value of  $x_t$  can be now computed without employing the external LP solver. Altogether, there are three actions  $a, b, c$  enabled in  $t$  resulting in the following three cases:

$$x_t^a = \frac{pv_v^a}{1-p_t^a} = \frac{0.7 \cdot 1}{1} = 0.7 \quad x_t^b = \frac{pv_u^b}{1-p_t^b} = \frac{0.3 \cdot 0.9}{1-0.7} = 0.9 \quad x_t^c = \frac{0}{1-0.5} = 0$$

Finally,  $x_t = \max(x_t^a, x_t^b, x_t^c) = 0.9$ .

### 3.4 Parallelization

The improved algorithm, described as Algorithm 3, consists of several consecutive phases, each of them parallelized to a certain level.

Parallel version of BdA algorithm performs qualitative model checking and computes  $AS$  as a basis for quantitative verification. The main idea builds on the topological sort for cycle detection – an algorithm that does not depend on DFS postorder and can be thus parallelized reasonably well. Minimal Subgraph Identification employs only one forward and one backward reachability and thus this phase is parallelized effectively. In order to parallelize SCC Decomposition,

---

**Algorithm 3** Improved algorithm for local quantitative analysis

---

```
1: compute the set  $AS$  using parallel version of BdA algorithm
2: compute the minimal subgraph  $MG$  using parallel reachability
3: decompose  $MG$  into SCCs using parallel OBF algorithm
4: build the quotient graph of  $MG$  in parallel
5: while there is an unsolved SCC do
6:   compute the set  $P$  of prepared SCCs
7:   for all  $C \in P$  do
8:     in parallel do
9:       if  $C$  is trivial then
10:        compute the solution of  $C$ 
11:       else
12:        create the linear programming problem  $LP_C$ 
13:        substitute for  $x_s$  such that  $s \in Output(C)$  in  $LP_C$ 
14:        compute the solution of  $LP_C$ 
15:       end if
16:       mark  $C$  as solved
17:     end in parallel
18:   end for
19: end while
20: return  $x_{init}$ 
```

---

the implementation is based on recursive variant of OBF algorithm as described in [6]. Iterative Computation allows to solve prepared SCCs independently by parallel running threads. However, each component has to be solved by calling the external serial LP solver `lpsolve`. This last limitation could be eventually relaxed by a parallel LP solver (we were unfortunately not able to get access to a suitable free parallel solver).

## 4 Experimental Evaluation

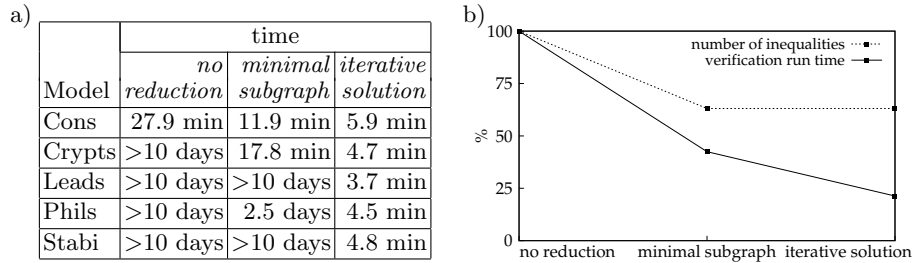
We have implemented all the described algorithms and techniques in the tool called PROBDIVINE. The tool uses DIVINE LIBRARY [20] and a generally available LP solver `lpsolve`. We ran a set of experiments on machines equipped with Intel Xeon 5130 and AMD Opteron 885 processors allowing us to measure the performance of the tool when using 1 to 8 threads.

We have used five different experimental models of randomized protocols with properties yielding minimal probability other than 0 or 1:

- Cons – randomized consensus protocol [2]
- Crypts – randomized dining cryptographers [9]
- Leads – asynchronous leader election protocol [18]
- Phils – randomized dining philosophers [19]
- Stabi – randomized self-stabilizing protocol [17]

| Model  | # states  | # inequalities for LP solver |                      |                        | % of the whole graph |                        |
|--------|-----------|------------------------------|----------------------|------------------------|----------------------|------------------------|
|        |           | <i>whole graph</i>           | <i>reduced graph</i> | <i>largest problem</i> | <i>reduced graph</i> | <i>largest problem</i> |
| Cons   | 48 669    | 132 243                      | 83 395               | 20 368                 | 63.06                | 15.40                  |
| Crypts | 2 951 903 | 8 954 217                    | 108 045              | 0                      | 1.21                 | 0                      |
| Leads  | 2 995 379 | 8 800 096                    | 5 678 656            | 0                      | 64.5                 | 0                      |
| Phils  | 5 967 065 | 14 740 726                   | 1 623 722            | 246                    | 11.0                 | Almost 0               |
| Stabi  | 4 061 570 | 6 897 480                    | 5 983 080            | 0                      | 86.7                 | 0                      |

**Table 1.** The size of LP problem with respect to used reduction techniques.



**Fig. 4.** a) Runtimes for various models when no reduction is used, when only the minimal subgraph is considered, and when both the minimal subgraph and iterative processing is involved. b) Correlation between the number of inequalities and runtime.

Table 1 captures the size of the linear programming problem (the number of inequalities to be solved by an external LP solver). The column *whole graph* gives the size before applying any reduction, the column *reduced graph* gives the size when redundant inequalities were removed by pruning the graph into the minimal subgraph. The column *largest problem* gives the maximal size of a problem to be solved by an LP solver, when the original problem was decomposed into subproblems that were processed independently. Three of the models contain only trivial SCCs, thus the LP solver is not called at all and the size of the largest problem solved by LP solver is thus 0.

The table in Figure 4 demonstrates that the size and the structure of the problem plays a crucial role in the performance of the tool. The table gives overall run times corresponding to the used reduction techniques. The first column gives run time when no reduction technique is used (*no reduction*), the second one when redundant inequalities are removed (*minimal subgraph*), and the third one when the technique of iterative computation and trivial SC solving are applied on the minimal subgraph (*iterative solution*). A correlation between times in Figure 4 and sizes in Table 1 is observable. With decreasing number of inequalities the runtimes tend to speed-up dramatically. As for the speed-up, the most

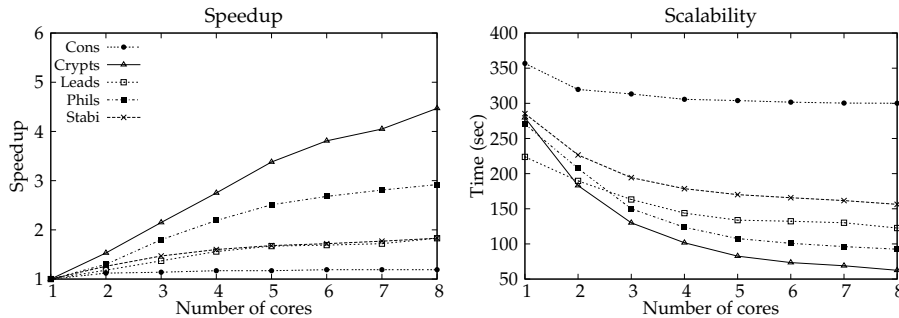


Fig. 5. Overall speed-up and scalability.

inconvenient case is when the graph is made of one large component. In such a case, the pruning and parallel processing cannot be done and the verification runtime is dominated by the single call to the LP solver.

In the case of *Cons* model the reduction techniques help less than in the other cases. However, run times decrease still significantly. Figure 4.a) gives runtimes of verification when various degree of improvement is used. Figure 4.b) depicts the relative decrease in the runtime and number of inequalities when various reduction techniques are involved. In particular, the number of inequalities decreases to 63% of the original number, while the runtime decreases to 21% of the original time needed to perform the verification task.

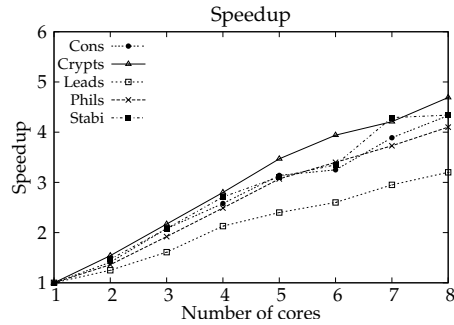
Figure 5 reports on the overall speed-up and scalability of the verification process we achieved using our tool on various number of CPU cores. Poor scalability in case of *randomized consensus protocol* can be explained, because the time consumed by the sequential LP solver takes the major part of the runtime of the whole verification process.

Figure 6 aims on the qualitative analysis as a part of the whole verification process. The table in Figure 6 shows the ratio between runtimes of the qualitative analysis and the whole verification process. The graph in Figure 6 presents speed-up of qualitative analysis (the first phase of the algorithm). In comparison to the quantitative verification, the speed-up is much better due to the fact, that the whole verification process contains phases where parallelization does not help much as they require frequent synchronization. We can observe, that the bigger the part the qualitative analysis forms in the whole verification process, the better the overall scalability is. In case of *dining cryptographers protocol* model, the qualitative verification forms 98.75 % of the whole verification process and all the LPs are solved without using external LP solver. Therefore the scalability and speed-up are the best over all the examples.

All in all, we claim that our approach is quite successful as overall runtimes tend to decrease as more CPU cores are used.

The structure of a graph is a crucial aspect affecting the runtime of the verification process. For instance the *Crypts* model contains approximately the same number of states as *Leads*, but runtimes of qualitative verification differ

| Model  | <i>qualitative analysis</i> | <i>whole analysis</i> | <i>% qualit of whole</i> |
|--------|-----------------------------|-----------------------|--------------------------|
| Cons   | 30.53                       | 358.15                | 8.52                     |
| Crypts | 275.96                      | 279.47                | 98.75                    |
| Leads  | 68.28                       | 223.76                | 30.51                    |
| Phils  | 180.6                       | 270.46                | 66.77                    |
| Stabi  | 67.36                       | 285.34                | 23.61                    |



**Fig. 6.** Ratio between runtimes of the qualitative analysis and the whole verification process (table on the left). Speed-up of qualitative analysis only (on the right).

a lot. On the other hand, runtime of *Leads* is comparable to *Stabi*, but their number of states and speedup differ.

## 5 Conclusion

As probabilistic systems gain popularity and are coming into wider use, the need for formal verification and analysis methods, techniques and tools capable of handling these systems become more critical. The theory and algorithms for formal verification of probabilistic systems have been around for some time. However, it is the existence of a good and efficient formal verification tool that makes the theory valid from the practitioner’s point of view.

In this paper we presented several techniques that allow to build competitive enumerative model checking tool for quantitative analysis of linear temporal properties over finite state probabilistic systems. In particular, we showed how to involve parallelism and employ locality to increase the performance of such a tool. We also showed that the costly call to the linear programming solver can be either replaced with multiple successive calls for smaller problems, or avoided at all. Using this approach we achieved order-of-magnitude reduction in runtime of verification in many cases.

## References

1. A. Aziz, K. Sanwal, V. Singhal, and R. K. Brayton. Verifying continuous-time Markov Chains. In *Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *LNCS*, pages 269–276. Springer Verlag, 1996.
2. J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 15(1):441–460, 1990.
3. C. Baier. On the Algorithmic Verification of Probabilistic Systems. Habilitation Thesis, Universität Mannheim, 1998.

4. C. Baier, M. Größer, and F. Ciesinski. Partial Order Reduction for Probabilistic Systems. In *1st International Conference on Quantitative Evaluation of Systems (QEST 2004)*, pages 230–239. IEEE Computer Society, 2004.
5. J. Barnat, L. Brim, I. Černá, M. Češka, and J. Tůmová. ProbDiVinE-MC: Multi-Core LTL Model Checker for Probabilistic Systems. In *Proceedings of QEST'08, Tool Paper, To appear*. IEEE, 2008, <http://anna.fi.muni.cz/probdivine>.
6. J. Barnat, J. Chaloupka, and J. van de Pol. Improved Distributed Algorithms for SCC Decomposition. *Electron. Notes Theor. Comput. Sci.*, 198(1):63–77, 2008.
7. A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In *Proc. FST&TCS 1995*, volume 1026 of *LNCS*, pages 499 – 513, 1995.
8. B. Jeannot, P. de Argenio, and K.G. Larsen. RAPTURE: A tool for verifying Markov Decision Processes. In *Proc. Tools Day / CONCUR'02. Tech. Rep. FIMU-RS-2002-05*, pages 84–98. MU Brno, 2002.
9. D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1:65–75, 1988.
10. F. Ciesinski and C. Baier. LiQuor: A tool for Qualitative and Quantitative Linear Time analysis of Reactive Systems. In *Proc. of QEST'06*, pages 131–132. IEEE Computer Society, 2006.
11. C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *Journal of the ACM*, 42(4):857–907, 1995.
12. L. de Alfaro. *Formal Verification of Stochastic Systems*. PhD thesis, Stanford University, Department of Computer Science, 1997.
13. C. Derman. *Finite State Markovian Decision Processes*. Academic Press, Inc., Orlando, FL, USA, 1970.
14. J.L. Doob. *Measure theory*. Springer-Verlag, 1994.
15. Hans Hansson and Bengt Jonsson. A Framework for Reasoning about Time and Reliability. In *IEEE Real-Time Systems Symposium*, pages 102–111, 1989.
16. A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In *Proc. of TACAS'06*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.
17. A. Israeli and M. Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 119–131, 1990.
18. A. Itai and M. Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1), 1990.
19. D. Lehmann and M. Rabin. On the advantage of free choice: A symmetric and fully distributed solution to the dining philosophers problem (extended abstract). In *Proc. 8th Annual ACM Symposium on Principles of Programming Languages (POPL'81)*, pages 133–138, 1981.
20. ProbDiVinE homepage. <http://anna.fi.muni.cz/probdivine> (2008).
21. M. L. Puterman. *Markov Decision Processes-Discrete Stochastic Dynamic Programming*. John Wiley & Sons, New York, 1994.
22. M.Y. Vardi. Probabilistic linear-time model checking: an overview of the automata-theoretic approach. In *Proc. Formal Methods for Real-Time and Probabilistic Systems, ARTS 1999*, volume 1601 of *LNCS*, pages 265–276. Springer, 1999.
23. M.Y. Vardi and P. Wolper. Reasoning about infinite computation paths. *Proceedings of 24th IEEE Symposium on Foundation of Computer Science, Tuscan*, pages 185–194, 1983.