On-the-fly Probabilistic Model Checking

Diego Latella
ISTI - CNR

Michele Loreti
Università di Firenze

Mieke Massink
ISTI - CNR

Model checking approaches can be divided into two broad categories: global approaches that determine the set of all states in a model $\mathscr{M}$ that satisfy a temporal logic formula Φ, and local approaches in which, given a state s in $\mathscr{M}$, the procedure determines whether s satisfies Φ. When s is a term of a process language, the model-checking procedure can be executed "on-the-fly", driven by the syntactical structure of s. For certain classes of systems, e.g. those composed of many parallel components, the local approach is preferable because, depending on the specific property, it may be sufficient to generate and inspect only a relatively small part of the state space. We propose an efficient, on-the-fly, PCTL model checking procedure that is parametric with respect to the semantic interpretation of the language. The procedure comprises both bounded and unbounded until modalities. The correctness of the procedure is shown and its efficiency is compared with a global PCTL model checker on representative applications.

1 Introduction and Related Work

Model checking approaches are often divided into two broad categories: *global* approaches that determine the set of all states in a model $\mathscr{M}$ that satisfy a temporal logic formula Φ, and *local* approaches in which, given a state s in $\mathscr{M}$, the procedure determines whether s satisfies Φ [5, 3]. When s is a term of a process language, the model checking procedure can be executed "on-the-fly", driven by the syntactical structure of s. On-the-fly algorithms are following a *top-down* approach that does not require global knowledge of the complete state space. For each state that is encountered, starting from a given state, the outgoing transitions are followed to adjacent states, constructing step by step local knowledge of the state space until it is possible to decide whether the given state satisfies the formula (or memory bounds are reached). Global algorithms instead, construct the set of states that satisfy a formula recursively in a *bottom-up* fashion following the syntactic structure of the formula [4] and require the full state space of the model to be generated before they can be applied.

In this paper, we present a local, on-the-fly, probabilistic model checking algorithm for full *Probabilistic Computation Tree Logic* (PCTL) [9], a probabilistic extension of the temporal logic *Computation Tree Logic* (CTL) [4] that includes both the bounded and unbounded until operator. The algorithm is *parametric* with respect to the semantic interpretation of the front-end language. Each instantiation of the algorithm consists of the appropriate definition of two functions: next and lab_eval. Function next, given a process term (or state)[1], returns a list of pairs. Each pair consists of a process term, that can be reached in one step from the given process term, and its related probability. Function lab_eval, given a term, gives a boolean function associating true to each atomic proposition with which the term is labelled. This parametric approach has the advantage that the model checker can be easily instantiated on specification languages with different semantics. For example, in [14] we present two different interpretations for bounded PCTL; one being the standard, exact probabilistic semantics of a simple, time-synchronous population description language, and the other being the mean-field approximation in discrete time of

[1] We will consider process terms as states throughout this paper.

I. Lanese, A. Lluch Lafuente, A. Sokolova, H. T. Vieira (Eds.):
7th Interaction and Concurrency Experience (ICE 2014)
EPTCS 166, 2014, pp. 45–59, doi:10.4204/EPTCS.166.6

such a semantics [16, 14, 15]. The mean-field approximation has proved a successful technique to analyse properties of individual components in the context of large population models in the discrete time setting[2]. The main contributions of the current paper are twofold: 1) we provide a detailed description of the on-the-fly algorithm (not presented in [14, 15]) together with the proofs of correctness. In particular, the algorithm for the unbounded until operator uses a new technique exploiting an interesting property of transient Discrete Time Markov Chains (DTMCs), i.e. one in which all recurrent states are absorbing; 2) we use an instantiation of the prototype on-the-fly PCTL model checker FlyFast on an automata based language and semantics such as that used in the PRISM model checker which in turn is used to make a first comparison for what concerns the efficiency of the on-the-fly algorithm when used for full PCTL model checking to make sure that its efficiency is at least comparable with PRISM in the worst case in which the whole state space must be explored.

Related work. In the context of qualitative model checking of temporal logics such as CTL [4], LTL [18, 19] and CTL*[3], local model checking algorithms have been proposed to mitigate the state space explosion problem using an on-the-fly approach [5, 3, 11, 7]. The have also the same worst-case complexity as the best existing global procedures for the above mentioned logics. However, they have better performance when only a subset of the system states need to be analysed to determine whether a system satisfies a formula. Such cases occur frequently in practice. Furthermore, local model checking may provide results for infinite state spaces.

In the context of probabilistic and stochastic model checking global algorithms have been more popular than local ones and can be found in many sophisticated tools such as PRISM, MRMC and many others [2, 13]. A clear advantage of these global algorithms is that results are obtained for *all* states of the model, if the state space is not too large, and that, depending on the particular formula to verify, usually the underlying model can be reduced to fewer states before the algorithm is applied and can be reduced to combinations of existing well-known and optimised algorithms for Markov chains such as transient analysis [2]. In the context of Markov Decision Processes (MDP) partial order reduction techniques have been explored to obtain state space reduction [6]. This technique is based on a static partial order reduction approach that, starting from the complete state space representation, produces an equivalent and compact representation of the state space that can be used as input of the model checking algorithm [1].

To the best of our knowledge the only algorithm for on-the-fly model checking for probabilistic processes is the one proposed in [17] that only considers the fragment of the PCTL without unbounded until. On the contrary, the local model-checking algorithm considered in this paper considers all PCTL. This is an important point. Indeed, the use of full PCTL forbids the application of specific techniques, like for instance statistical model checking, e.g. [20], that can be used only when computations with a bounded temporal horizontal.

2 Probabilistic Computation Tree Logic

In this section we briefly recall the definition of the *Probabilistic Computation Tree Logic* (PCTL) [9], a probabilistic extension of the temporal logic CTL [4], for the expression of properties of Discrete Time Markov Chains (DTMCs) and Markov Decision Processes (MDPs). The syntax of PCTL is the

[2]The mean-field technique approximates the mean global behaviour of the population by a deterministic limit that provides at each time step the expected number of objects that are in the various local states. The iterative calculation of the mean-field in combination with the process modelling the single object forms a DTMC that lends itself very well to on-the-fly analysis and the computational complexity is insensitive to the size of the population.

$s \models_{\mathscr{M}} a$	iff	$a \in \ell(s)$
$s \models_{\mathscr{M}} \neg\Phi$	iff	not $s \models_{\mathscr{M}} \Phi$
$s \models_{\mathscr{M}} \Phi_1 \vee \Phi_2$	iff	$s \models_{\mathscr{M}} \Phi_1$ or $s \models_{\mathscr{M}} \Phi_2$
$s \models_{\mathscr{M}} \mathscr{P}_{\bowtie p}(\varphi)$	iff	$\mathbb{P}\{\sigma \in Paths_{\mathscr{M}}(s) \mid \sigma \models_{\mathscr{M}} \varphi\} \bowtie p$
$\sigma \models_{\mathscr{M}} \mathscr{X}\ \Phi$	iff	$\sigma[1] \models_{\mathscr{M}} \Phi$
$\sigma \models_{\mathscr{M}} \Phi_1\, \mathscr{U}^{\leq k}\, \Phi_2$	iff	$\exists 0 \leq h \leq k\ s.t.\ \sigma[h] \models_{\mathscr{M}} \Phi_2 \wedge \forall 0 \leq i < h\ .\ \sigma[i] \models_{\mathscr{M}} \Phi_1$
$\sigma \models_{\mathscr{M}} \Phi_1\, \mathscr{U}\, \Phi_2$	iff	$\exists 0 \leq k\ s.t.\ \sigma[k] \models_{\mathscr{M}} \Phi_2 \wedge \forall 0 \leq i < k\ .\ \sigma[i] \models_{\mathscr{M}} \Phi_1$

Table 1: Satisfaction relation for PCTL.

following:

$$\Phi ::= a \mid \neg\Phi \mid \Phi \vee \Phi \mid \mathscr{P}_{\bowtie p}(\varphi) \qquad \text{where } \varphi ::= \mathscr{X}\,\Phi \mid \Phi\,\mathscr{U}^{\leq k}\,\Phi \mid \Phi\,\mathscr{U}\,\Phi$$

where $a \in \mathscr{P}$ is an atomic proposition, $\bowtie \in \{\leq, <, >, \geq\}$, $p \in [0,1]$ and $k \in \mathbb{N}$. PCTL formulas are interpreted over *state labelled* DTMCs and consist of all the state formulas Φ. The path formulas φ only appear as parameter of the operator $\mathscr{P}_{\bowtie p}(\varphi)$. Informally, a state s in a DTMC satisfies $\mathscr{P}_{\bowtie p}[\varphi]$ if the total probability measure of the set of paths that satisfy path formula φ is $\bowtie p$. A state labelled DTMC is a pair $\langle \mathscr{M}, \ell \rangle$ where $\mathscr{M}$ is a DTMC with state set $\mathscr{S}$ and $\ell : \mathscr{S} \to 2^{\mathscr{P}}$ associates each state with a set of atomic propositions; for each state $s \in \mathscr{S}$, $\ell(s)$ is the set of atomic propositions true in s. In the following, we assume $\mathbf{P}$ be the one step probability matrix for $\mathscr{M}$; we abbreviate $\langle \mathscr{M}, \ell \rangle$ with $\mathscr{M}$, when no confusion can arise. A path σ over $\mathscr{M}$ is a non-empty sequence of states $s_0, s_1, \cdots$ where $\mathbf{P}_{s_i, s_{i+1}} > 0$ for all $i \geq 0$. We let $Paths_{\mathscr{M}}(s)$ denote the set of all infinite paths over $\mathscr{M}$ starting from state s. By $\sigma[i]$ we denote the i-th element s_i of path σ, for $i \geq 0$. The satisfaction relation on $\mathscr{M}$ and the logic are formally defined in Table 1. For every path formula φ, the set $\{\sigma \in Paths_{\mathscr{M}}(s) | \sigma \models \varphi\}$ is a *measurable set* [13].

3 On-the-fly Probabilistic Model Checking

We introduce a local on-the-fly model checking algorithm for PCTL on labeled DTMC $\langle \mathscr{M}, \ell \rangle$. The basic idea of an on-the-fly algorithm is simple: while the state space is generated in a stepwise fashion from a term s of the language, the algorithm keeps track of all the paths that are being generated. For each of them it updates the information about the satisfaction of the formula that is checked. In this way, only that part of the state space is generated that may provide information on the satisfaction of the formula and irrelevant parts are not taken into consideration, mitigating the problem of *state space* explosion. However, the proposed model checking algorithm is not only based on graph generation. Indeed, while the relevant part of the state space is generated, the satisfaction probabilities of path formulas are also computed (on-the-fly).

The proposed algorithm abstracts from any specific language and from different semantic interpretations of a language. We only assume an abstract interpreter function that, given a generic process term, returns a probability distribution over the set of terms. Below, we let proc be the (generic) type of *probabilistic process terms*[3] while we let formula and path_formula be the types of *state-* and *path-* PCTL formulas. Finally, lab denotes the type of *atomic propositions*.

The abstract interpreter can be modelled by means of two functions: next and lab_eval. Function next associates a list of pairs $(\mathsf{proc}, \mathsf{float})$ to each element of type proc. The list of pairs gives the

[3]

```
1  boolean Check( s:proc, Φ:formula ) {
2    switch (Φ) {
3      case a: return lab_eval(s,a);
4      case ¬Φ1: return ¬Check( s , Φ1 );
5      case Φ1∨Φ2: return Check( s , Φ1 ) ∨ Check( s , Φ2 );
6      case 𝒫⋈p(φ): return CheckPath(s,φ) ⋈ p;
7    }
8  }
```

Table 2: Function Check

```
1  float CheckPath( s:proc, φ:path_formula ) {
2   switch φ {
3    case 𝒳Φ: {
4      p = 0.0;
5      lst = next(s);
6      for (s′,p′) ∈ lst {
7       if (Check(s′,Φ)) { p = p+p′;}
8      }
9      return p;
10   }
11   case Φ1 𝒰^≤k Φ2: return CheckBoundedUntil( s , Φ1 , k , Φ2 );
12   case Φ1 𝒰 Φ2: return CheckUnboundedUntil( s , Φ1 , Φ2 );
13  }
14 }
```

Table 3: Function CheckPath

terms, i.e. states, that can be reached in one step from the given state and their one-step transition probability. We require that for each s of type proc it holds that $0 < p' \leq 1$, for all $(s', p') \in \mathsf{next}(s)$ and $\sum_{(s',p')\in \mathsf{next}(s)} p' = 1$. Function lab_eval returns for each element of type proc a function associating a bool to each atomic proposition a in lab. Each instantiation of the algorithm consists in the appropriate definition of next and lab_eval, depending on the language at hand and its semantics.

The local model checking algorithm is defined as a function, Check, shown in Table 2. On atomic state-formulas, the function returns the value of lab_eval; when given a non-atomic state-formula, Check calls itself recursively on sub-formulas, in case they are state-formulas, whereas it calls function CheckPath, in case the sub-formula is a path-formula. In both cases the result is a Boolean value that indicates whether the state satisfies the formula[4].

Function CheckPath, shown in Table 3, takes two input parameters: a state $s \in$ proc and a PCTL path-formula $\varphi \in$ path_formula. As a result, it produces the probability measure of the set of paths, starting in state s, which satisfy path-formula φ. Following the definition of the formal semantics of PCTL, three different cases can be distinguished. If $\varphi = \mathscr{X}\,\Phi$ then the result is the sum of the probabilities of the transitions from s to those next states s' that satisfy Φ. To verify the latter, function Check is recursively invoked on such states. If φ is $\Phi_1\,\mathscr{U}^{\leq k}\,\Phi_2$ or $\Phi_1\,\mathscr{U}\,\Phi_2$ functions CheckBoundedUntil or CheckUnboundedUntil are invoked, respectively. These functions are presented in the next two subsections.

Let s be a term of a probabilistic process language and $\mathscr{M}$ the complete discrete time stochastic process associated with s by the formal semantics of the language.

Theorem 3.1 *$s \models_{\mathscr{M}} \Phi$ if and only if Check$(s,\Phi) =$ true.*

Proof. The theorem is proven by induction on the structure of Φ. The more involved parts concern the proof of the theorem for the path formulas concerning bounded and unbounded until. These are provided

[4]For obvious reasons of presentation here we show a simplified, not fully optimised, pseudo-code version of the algorithm.

```
1  float CheckBoundedUntil(s:proc , Φ1:formula , k:int , Φ2:formula) {
2   r = createBUStructure( s , Φ1 , k , Φ2 );
3   M = [s↦r];
4   if (r.label == YES) { return 1.0; }
5   if (r.label == NO) { return 0.0; }
6   S_yes = ∅;
7   toExpand = {r};
8   c = k;
9   while (c > 0) ∧ (toExpand ≠ ∅) {
10   T = toExpand;
11   toExpand = ∅;
12   for (r ∈ T) {
13    lst = next(r.term);
14    for (s', p') ∈ lst {
15     r'=M[s'];
16     if (r' == ⊥) {
17      r' = createBUStructure( s' , Φ1 , k , Φ2 );
18      M = M[s' ↦ r'];
19      if (r'.label == YES) {
20       S_yes= S_yes ∪ {r'};
21      } else if (r'.label != NO) {
22       toExpand = toExpand ∪ {r'};
23      }
24     }
25     r'.prec = (r,p) :: r'.prec;
26    }
27   }
28   c = c−1;
29  }
30  if (S_yes == ∅) { return 0.0; }
31  A = S_yes;
32  for (i = 1; i <= k; i++) {
33   for (r ∈ A) {
34    for ((r', p') ∈ r.prec) {
35     r'.p[i] = r'.p[i] + p' ∗ r.p[i−1];
36    }
37   }
38   A = {r | ∃r' ∈ A : r ⪯ r'};
39  }
40  return r.p[k];
41 }
```

Table 4: Function CheckBoundedUntil

as Lemma 3.1, Lemma 3.2 and Lemma 3.4 together with an outline of their proofs in the following sections.

□

3.1 Computing Bounded Until Probability

Function CheckBoundedUntil, defined in Table 4, computes the probability of the set of paths starting from a given state s that satisfy formula $\Phi_1 \mathscr{U}^{\leq k} \Phi_2$. This function takes as parameters a state s, state formulas Φ_1 and Φ_2, and the bound k. Notice that differently from the algorithm proposed in [17], where a recursive algorithm is proposed, to compute the probability of path satisfying $\Phi_1 \mathscr{U}^{\leq k} \Phi_2$ an iterative solution is proposed. Moreover, the proposed procedure is not an adaptation of standard PCTL where all the state space is considered to compute the requested probability value.

To compute $\mathbb{P}\{\sigma \in \mathit{Paths}_{\mathscr{M}}(s) \mid \sigma \models_{\mathscr{M}} \Phi_1 \mathscr{U}^{\leq k} \Phi_2\}$, function CheckBoundedUntil first populates a data structure M with states reachable from s in at most k steps (lines 1–28). We refer to this phase as the *expansion* phase[5].

The use of this data structure enables *memoization* and permits reusing the probability values com-

[5] A similar approach is used in [8] to analyse infinite Markov chains. However in [8] after the expansion phase (that is used to compute a finite truncation of the original system), the standard model checking algorithm is used. Indeed, differently from the solution proposed in this paper, satisfaction of Φ_1 and Φ_2 does not play any rôle.

```
1  BURecord createBUStructure(s : proc , Φ1 : formula , k : int , Φ2 : formula) {
2   l = UNKNOWN;
3   p = new float[k+1];
4   if (Check( s , Φ2 )) {
5    l = YES;
6    ∀0 ≤ i ≤ k.p[i] = 1.0;
7   } else if (¬Check( s , Φ1 )) {
8    l = NO;
9   }
10  return ⟨term = s; prec = []; p = p; label = l⟩;
11 }
```

Table 5: Function createBUStructure

puted already computed in different sub-formulae. Structure M is a *hashmap*[6] that associates each (reachable) process term s' with a record of type BURecord with the following fields:

- term: a value of type proc referring to the associated process term s'.
- prec: a list of *predecessors* consisting of pairs $(\mathsf{BURecord}, \mathsf{float})$. Intuitively, given a BURecord r, a pair $(\mathsf{r}', \mathsf{p}')$ occurs in r.prec if and only if r'.term evolves in one step to r.term with probability p' and has a record in M.
- p: a float array of probabilities. The $i-th$ element in the array, p[i], will contain $\mathbb{P}\{\sigma \in \mathit{Paths}_{\mathscr{M}}(s') \mid \sigma \models_{\mathscr{M}} \Phi_1 \mathscr{U}^{\leq i} \Phi_2\}$[7].
- label: a label taking a value in $\{\mathsf{YES}, \mathsf{NO}, \mathsf{UNKNOWN}\}$. This field takes value YES when term satisfies Φ_2. When term satisfies neither Φ_2 nor Φ_1 the field label takes value NO and when it satisfies only Φ_1 it takes value UNKNOWN.

We introduce the record precedence relation $\prec$. Let r and r' be two BURecord, we write $r \prec r'$ if and only if there exists probability $p > 0$ such that $(r, p) \in r'.\mathsf{prec}$. We will also use $r \preceq r'$ to denote that either $r = r'$ or $r \prec r'$ and $\preceq^i$ for i-steps precedence.

CheckBoundedUntil uses function createBUStructure to allocate new instances of BURecord for the starting state s and further relevant states that are reachable from s. This function, defined in Table 5, takes as parameter a state s, two state formulas Φ_1 and Φ_2 and the bound k. The label field of the returned record is initialised to YES, NO or UNKNOWN as above by means of further calls of function Check. CheckBoundedUntil initially checks the label for its parameter s (lines 4 and 5), if such label is either YES or NO the values 1.0 or 0.0, respectively, are returned and there is no need to continue expansion. Otherwise the actual expansion phase is entered (lines 6-29). For each state to be expanded, the list of states reachable in one step from such a state is computed using function next; a new record r' is created for each state s' in the list which does not appear already in M. During this phase the set *toExpand* is used to keep record of those r' which still need to be expanded, whereas set S_{yes} collects all r' representing states which satisfy Φ_2, i.e. are labelled YES. Furthermore, the list of predecessors of r' is updated accordingly. Additional predecessors are added also when r' was already inserted in M because it was visited before.

When the expansion phase is completed, function CheckBoundedUntil tests whether S_{yes} is empty (line 30, Table 4). In that case value 0.0 is returned because no state satisfying Φ_2 can be reached from s within k steps. Hence, $\mathbb{P}\{\sigma \in \mathit{Paths}_{\mathscr{M}}(s) \mid \sigma \models_{\mathscr{M}} \Phi_1 \mathscr{U}^{\leq k} \Phi_2\}$ is 0.0. If $S_{yes} \neq \emptyset$, function

[6] In this paper we use $\{\}$ to denote the empty hashmap, while $\mathsf{M}[\mathsf{x} \mapsto \mathsf{y}]$ denotes the hashmap obtained from M by adding the association of *value* y to *key* x. We also use $\{\mathsf{x} \mapsto \mathsf{y}\}$ to denote $\{\}[\mathsf{x} \mapsto \mathsf{y}]$.

[7] For the sake of readability, we explicitly consider all the components occurring in the array. When the algorithm is implemented, we do not need to store the whole array explicitly.

CheckBoundedUntil enters the *computation phase* (lines 32–39). This phase starts from YES-labelled records (now stored in variable A, indicating the *active* records). Then, the probability to reach a YES-labelled node within i steps is iteratively computed in a backward fashion (i ranging from 1 to k). Note that a state could be the predecessor of more than one state that is on a path to a YES-labelled state. This is why the probability of a state to reach a YES-labelled state in at most i steps is the sum of the probability accumulated due to being a predecessor of other states and the probability due to being a predecessor of the currently considered state in A. The total probability mass is obtained when the maximal number of steps k is reached. At the end of each iteration, the set A is updated by considering further states directly preceding those currently in A, i.e. A is updated as follows: $\{r | \exists r' \in A : r \preceq r'\}$. After i iterations, the set A contains all the states in M that can reach an element in S_{yes} in at most i steps.

Lemma 3.1 *For each s, Φ_1, k, and Φ_2, let CheckBoundedUntil$(s,\Phi_1,k,\Phi_2) = p$ and* M *be the data structure obtained at the end of the expansion phase, one of the following holds:*

1. $\mathsf{M}[s].label = \mathsf{YES}$ *and* $p = 1.0$;
2. $\mathsf{M}[s].label = \mathsf{NO}$ *and* $p = 0.0$;
3. $\mathsf{M}[s].label = \mathsf{UNKNOWN}$, $S_{yes} = \emptyset$ *and* $p = 0.0$;
4. $\mathsf{M}[s].label = \mathsf{UNKNOWN}$, $S_{yes} \neq \emptyset$ *and*

$$p = \mathbb{P}\{\sigma \in Paths_{\mathscr{M}}(s) \mid \exists i \leq k.\mathsf{M}[\sigma[i]].label = \mathsf{YES} \wedge \forall j < i.\mathsf{M}[\sigma[j]].label = \mathsf{UNKNOWN}\}$$

Proof. If CheckBoundedUntil terminates its execution at line 4, 5 or 30, then the first three cases are readily proven. If CheckBoundedUntil terminates at line 40, the last case follows directly from the fact that at line 32 the following two loop invariants hold for iterations i ranging from 1 to k:

$$A = \{r' | \exists r'' \in S_{yes} : r' \preceq^i r''\}$$

$$\forall s'.\mathsf{M}[s'] = r \neq \bot, \forall j < i : r.p[j] = \mathbb{P}\{\sigma \in Paths^{\mathscr{R}(s,k)}_{\mathscr{M}}(r.term) \mid \exists i' \leq j.\mathsf{M}[\sigma[i']].label = \mathsf{YES} \wedge \forall j' < i'.\mathsf{M}[\sigma[j']].label = \mathsf{UNKNOWN}\}$$

where $\mathscr{R}(s,k)$ denotes the set of states s' that are reachable from s in at most k steps, while $Paths^{\mathscr{R}(s,k)}_{\mathscr{M}}(s')$ denotes the set of paths starting from s' that in the first k steps only pass through states in $\mathscr{R}(s,k)$. Note that the following equation is straightforward:

$$\mathbb{P}\{\sigma \in Paths^{\mathscr{R}(s,k)}_{\mathscr{M}}(s) \mid \exists i' \leq k.\mathsf{M}[\sigma[i']].label = \mathsf{YES} \wedge \forall j' < i'.\mathsf{M}[\sigma[j']].label = \mathsf{UNKNOWN}\} = \mathbb{P}\{\sigma \in Paths_{\mathscr{M}}(s) \mid \exists i' \leq k.\mathsf{M}[\sigma[i']].label = \mathsf{YES} \wedge \forall j' < i'.\mathsf{M}[\sigma[j']].label = \mathsf{UNKNOWN}\}$$

The proof of the Lemma follows directly from the two invariants and from the equation above. The correctness of the invariants is proven by induction on i. In the following, we will use A_i to denote the set A at iteration i.
Base of Induction: If $i = 1$ the statement follows directly from the fact that $A = S_{yes} \subset \mathscr{R}(s,k)$.
Induction Hypothesis: For each $i \leq n$ we have that at line 32 the following hold:

$A_i = \{r | \exists r'' \in S_{yes} : r \preceq^i r''\}$
$\forall s'.\mathsf{M}[s'] = r \neq \bot, \forall j < i :$
$r.p[j] = \mathbb{P}\{\sigma \in Paths_{\mathscr{M}}^{\mathscr{R}(s,k)}(r.term) \mid \exists i' \leq j.\mathsf{M}[\sigma[i']].label = \mathsf{YES} \wedge \forall j' < i'.\mathsf{M}[\sigma[j']].label = \mathsf{UNKNOWN}\}$

Inductive Step: Let us consider the case $i = n+1$. First of all, we have that:

$$\begin{array}{rcl} A_{n+1} & = & \{r | \exists r' \in A_n : r \preceq r'\} \\ & \stackrel{I.H.}{=} & \{r | \exists r' \exists r'' \in S_{yes} : r' \preceq^n r'' \wedge r \preceq r'\} \\ & = & \{r | \exists r'' \in S_{yes} : r \preceq^{n+1} r''\} \end{array}$$

Moreover, for each r such that there exists s: $\mathsf{M}[s] = r$ we have that (line 35):

$$r.p[n+1] = \sum_{\{r' | r' \in A_n \wedge (r,p') \in r'.prec\}} r'.p[n] * p'$$

By I.H., we have that:
$r'.p[n] = \mathbb{P}\{\sigma \in Paths_{\mathscr{M}}^{\mathscr{R}(s,k)}(r.term) \mid \exists i \leq n.\mathsf{M}[\sigma[i]].label = \mathsf{YES} \wedge \forall j < i.\mathsf{M}[\sigma[j]].label = \mathsf{UNKNOWN}\}$
Moreover, for each $r \notin A_n$, we have that $r.p[n] = 0.0$, proving that for each r:

$$r.p[n+1] = \mathbb{P}\{\sigma \in Paths_{\mathscr{M}}^{\mathscr{R}(s,k)}(r.term) \mid \exists i \leq n+1.\mathsf{M}[\sigma[i]].label = \mathsf{YES} \wedge \forall j < i.\mathsf{M}[\sigma[j]].label = \mathsf{UNKNOWN}\}$$

which proves the correctness of the two invariants.

□

3.2 Computing Unbounded Until Probability

In this section we present function CheckUnboundedUntil that can be used to compute the probability of the set of paths satisfying $\Phi_1 \mathscr{U} \Phi_2$ starting from a state s. Similarly to function CheckBoundedUntil considered in the previous section, function CheckUnboundedUntil, defined in Table 7, is structured in two phases: an *expansion phase* (lines 2–30) and a *computation phase* (lines 35-48). In the first phase, all the process terms reachable from state s, that are relevant for the computation of $\mathbb{P}\{\sigma \in Paths_{\mathscr{M}}(s) \mid \sigma \models_{\mathscr{M}} \Phi_1 \mathscr{U} \Phi_2\}$, are generated. The discovered terms are stored in a hash map M associating each reachable process term with an instance of record UURecord. This record type is the same as BURecord except for the field p which is replaced by two arrays of float elements p_{yes} and p_{no}. The role of these two fields will be clarified soon. Function createUStructure, defined in Table 6, is used to allocate a new instance of UURecord.

We use the same notation for the record precedence relation $\prec$ on UURecord as was introduced for BURecord. We will also use $\preceq^*$ to denote the transitive closure of $\preceq$. During the *expansion phase* (see Table 7, lines 9-30) the sets S_{yes} and S_{no} are populated. These sets eventually contain all the records that are labelled YES and NO, respectively. The expansion terminates when no new record is found. Before starting the *computation phase*, it is first checked whether S_{yes} is empty (Line 31 in Table 7) . If S_{yes} is empty then the value 0.0 is returned because no state satisfying Φ_2 can be reached starting from s. If S_{yes} is not empty, all the records that *cannot reach* YES-labelled records are added to S_{no}, their labels updated to NO and the probability p_{no} set to 1. If the resulting set S_{no} at this point is empty, the value 1.0 is returned because in this case s can only eventually reach YES-labelled records. If $S_{no} \neq \emptyset$, all the records that *cannot reach* NO-labelled records are added to S_{yes}, labelled by YES and their probability value

```
1  UURecord createUStructure(s : proc , Φ1 : formula , Φ2 : formula) {
2   l = UNKNOWN;
3   p_yes = new float[2];
4   p_no = new float[2];
5   if (Check( s , Φ2 )) {
6    l = YES;
7    p_yes = { 1.0 , 1.0 };
8    p_no = { 0.0 , 0.0 };
9   } else if (¬Check( s , Φ1 )) {
10   l = NO;
11   p_yes = { 0.0 , 0.0 };
12   p_no = { 1.0 , 1.0 };
13  }
14  return ⟨term = s; prec = []; p_yes = p_yes; p_no = p_no; label = l⟩;
15 }
```

Table 6: Function createUStructure

is set to 1.0. An example could be the occurrence of bottom strongly connected components (BSCC) consisting exclusively of states satisfying Φ_1. The states on such BSCCs cannot reach states that are labelled YES (or NO), but they can be treated as states labelled NO.

The *computation phase* (starting at line 39 in Table 7) operates on a set of *active* records A. Initially, A is $S_{yes} \cup S_{no}$. At the end of each iteration (line 42 in Table 7) A is extended with all the process terms that are able to reach elements in A in one step. At the end of iteration i, for each element r in A, $\mathsf{r.p_{yes}}[i \mod 2]$ contains the probability mass of the set of paths starting from r.term that reach within i steps a YES-labelled term while passing only through UNKNOWN-labelled records. Similarly, at the end of the same iteration i, $\mathsf{r.p_{no}}[i \mod 2]$ stores the probability of the set of paths starting from r.term that reach within i steps a NO-labelled term while only passing through UNKNOWN-labelled records. The computation phase terminates when, for each record r stored in M, the following holds: $\mathsf{r.p_{yes}}[i \mod 2] + \mathsf{r.p_{no}}[i \mod 2] \geq 1 - \varepsilon$, where ε is a given accuracy level. The reason for computing both $\mathsf{r.p_{yes}}$ and $\mathsf{r.p_{no}}$ is that this way a well-known property of transient DTMCs can be exploited to detect and predict when a sufficiently accurate result has been obtained and the computation can be terminated (see Sect. 3.3). Furthermore, only the current and next value of the probability needs to be stored. This is obtained by using two fields that are addressed via the index modulo 2. For the rest, the computation of the respective probabilities of p_{yes} and p_{no} follow the same pattern as for bounded until.

Lemma 3.2 *For each s, Φ_1 and Φ_2, let $\mathit{CheckUnboundedUntil}(s, \Phi_1, \Phi_2) = p$ and M be the data structure obtained at the end of the expansion phase, one of the following holds:*

1. $\mathsf{M}[s].\mathit{label} = \mathsf{YES}$ *and* $p = 1.0$;
2. $\mathsf{M}[s].\mathit{label} = \mathsf{NO}$ *and* $p = 0.0$;
3. $\mathsf{M}[s].\mathit{label} = \mathsf{UNKNOWN}$, $S_{yes} = \emptyset$ *and* $p = 0.0$;
4. $\mathsf{M}[s].\mathit{label} = \mathsf{UNKNOWN}$, $S_{no} = \emptyset$ *and* $p = 1.0$;
5. $\mathsf{M}[s].\mathit{label} = \mathsf{UNKNOWN}$, $S_{yes}, S_{no} \neq \emptyset$ *and*

$$|\mathbb{P}\{\sigma \in \mathit{Paths}_{\mathscr{M}}(s) \mid \exists i.\mathsf{M}[\sigma[i]].\mathit{label} = \mathsf{YES} \wedge \forall j < i.\mathsf{M}[\sigma[j]].\mathit{label} = \mathsf{UNKNOWN}\} - p| \leq \varepsilon$$

Proof. If CheckUnboundedUntil terminates its computation at line 4, 5, 31 or 34, then the first four cases apply respectively in a straightforward way. If CheckUnboundedUntil terminates at line 51, the statement follows directly from the fact that the following three invariants hold at line 39 for iteration i:

```
1  float CheckUnboundedUntil(s: proc , Φ1: formula , Φ2: formula) {
2   r = createUStructure( s , Φ1 , Φ2 );
3   M = {s ↦ r};
4   if (r.label == YES) { return 1.0; }
5   if (r.label == NO) { return 0.0; }
6   S_yes = ∅;
7   S_no = ∅;
8   toExpand = {r};
9   while (toExpand ≠ ∅) {
10   T = toExpand;
11   toExpand = ∅;
12   for (r ∈ T) {
13    lst = next(r.term);
14    for (s', p') ∈ lst {
15     r'=M[s'];
16     if (r' == ⊥) {
17      r' = createUStructure( s' , Φ1 , Φ2 );
18      M = M[s' ↦ r'];
19      if (r'.label == YES) {
20       S_yes= S_yes ∪ {r'};
21      } else if (r'.label == NO) {
22       S_no=S_no ∪ {r'};
23      } else {
24       toExpand = toExpand ∪ {r'};
25      }
26     }
27     r'.prec = (r,p) :: r'.prec;
28    }
29   }
30  }
31  if (S_yes == ∅) { return 0.0; }
32  S_no = {r| ∄r' ∈ S_yes.r ⪯* r'}
33  ∀r ∈ S_no.r.p_no = {1,1}, r.label = NO;
34  if (S_no == ∅) { return 1.0; }
35  S_yes = {r| ∄r' ∈ S_no.r ⪯* r'}
36  ∀r ∈ S_yes.r.p_yes = {1,1}, r.label = YES;
37  A = S_yes ∪ S_no;
38  i = 0;
39  while (∃(s,r) ∈ M : r.p_yes[i mod 2] + r.p_no[i mod 2] < 1 − ε) {
40    ∀r ∈ A.r.p_yes[(i+1) mod 2]) = 0;
41    ∀r ∈ A.r.p_no[(i+1) mod 2]) = 0;
42   for (r ∈ A) {
43    for ((r', p') ∈ r.prec) {
44     r'.p_yes[(i+1) mod 2] = r'.p_yes[(i+1) mod 2] + p' ∗ r.p_yes[i mod 2];
45     r'.p_no[(i+1) mod 2] = r'.p_no[(i+1) mod 2] + p' ∗ r.p_no[i mod 2];
46    }
47   }
48   i = i+1;
49   A = {r|∃r' ∈ A : r ⪯ r'};
50  }
51  return r.p[i mod 2];
52 }
```

Table 7: Function CheckUnboundedUntil

$$A = \{r' | \exists r'' \in S_{yes} \cup S_{no} : r' \preceq^i r''\}$$

$$\forall s'.\mathsf{M}[s'] = r \neq \bot,$$
$$r.p_{yes}[i \mod 2] = \mathbb{P}\{\sigma \in Paths_{\mathscr{M}}(r.term) \mid \exists i' \leq i.\mathsf{M}[\sigma[i']].label = \mathsf{YES} \wedge \forall j' < i'.\mathsf{M}[\sigma[j']].label = \mathsf{UNKNOWN}\}$$

$$\forall s'.\mathsf{M}[s'] = r \neq \bot,$$
$$r.p_{no}[i \mod 2] = \mathbb{P}\{\sigma \in Paths_{\mathscr{M}}(r.term) \mid \exists i' \leq i.\mathsf{M}[\sigma[i']].label = \mathsf{NO} \wedge \forall j' < i'.\mathsf{M}[\sigma[j']].label = \mathsf{UNKNOWN}\}$$

Let $\mathsf{CheckUnboundedUntil}(s, \Phi_1, \Phi_2) = p$ and $r = \mathsf{M}[s]$, then $p = r.p_{yes}[i \mod 2]$ and $1 \geq r.p_{yes}[i \mod 2] + r.p_{no}[i \mod 2] \geq 1 - \varepsilon = (p^s_{yes} + p^s_{no}) - \varepsilon$ where:

$$p^s_{yes} = \mathbb{P}\{\sigma \in Paths_{\mathscr{M}}(r.term) \mid \exists i : \mathsf{M}[\sigma[i]].label = \mathsf{YES} \wedge \forall j < i.\mathsf{M}[\sigma[j]].label = \mathsf{UNKNOWN}\}$$

$$p^s_{no} = \mathbb{P}\{\sigma \in Paths_{\mathscr{M}}(r.term) \mid \exists i.\mathsf{M}[\sigma[i]].label = \mathsf{NO} \wedge \forall j < i.\mathsf{M}[\sigma[j]].label = \mathsf{UNKNOWN}\}$$

Since $r.p_{yes}[i \mod 2] \leq p^s_{yes}$ and $r.p_{no}[i \mod 2] \leq p^s_{no}$, this implies that:

$$(p^s_{yes} - r.p_{yes}[i \mod 2]) + (p^s_{no} - r.p_{no}[i \mod 2]) - \varepsilon \leq 0$$

Hence: $p^s_{yes} - r.p_{yes}[i \mod 2] \leq \varepsilon$ which proves Lemma 3.2, apart from the three invariants, considered above, which are proven below. The proof of the first invariant is identical to the one considered in the proof of Lemma 3.1 and we omit it. The other two invariants are proven by induction on i. We only show the proof for the invariant concerning p_{yes}, the other being very similar.

Base of Induction: If $i = 1$ the statement follows directly from the fact that $A_0 = S_{yes} \cup S_{no}$, where we use A_i to denote the value of set A at iteration i. Note that for each $r \in M$, if $r \notin A_0$, $r.p_{yes}[i \mod 2] = r.p_{no}[i \mod 2] = 0.0$.

Induction Hypothesis: For each $i \leq n$ we have that at line 39 the following holds:

$$\begin{array}{l}\forall s'.\mathsf{M}[s'] = r \neq \bot,\\ r.p_{yes}[i \mod 2] = \mathbb{P}\{\sigma \in \mathit{Paths}_{\mathscr{M}}(r.\mathit{term}) \mid \exists i' \leq i.\mathsf{M}[\sigma[i']].\mathit{label} = \mathsf{YES} \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad \forall j' < i'.\mathsf{M}[\sigma[j']].\mathit{label} = \mathsf{UNKNOWN}\}\end{array}$$

$$\begin{array}{l}\forall s'.\mathsf{M}[s'] = r \neq \bot,\\ r.p_{no}[i \mod 2] = \mathbb{P}\{\sigma \in \mathit{Paths}_{\mathscr{M}}(r.\mathit{term}) \mid \exists i' \leq i.\mathsf{M}[\sigma[i']].\mathit{label} = \mathsf{NO} \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad \forall j' < i'.\mathsf{M}[\sigma[j']].\mathit{label} = \mathsf{UNKNOWN}\}\end{array}$$

Inductive Step: Let us consider the case $i = n+1$. For each r such that there exists s: $\mathsf{M}[s] = r$ we have that (lines 44-45 in Table 7):

$$r.p_{yes}[(n+1) \mod 2] = \sum_{\{r' \mid r' \in A_n \wedge (r,p') \in r'.\mathit{prec}\}} r'.p_{yes}[n \mod 2] * p'$$

$$r.p_{no}[(n+1) \mod 2] = \sum_{\{r' \mid r' \in A_n \wedge (r,p') \in r'.\mathit{prec}\}} r'.p_{no}[n \mod 2] * p'$$

By induction hypothesis, we have that:

$$\begin{array}{l}r'.p_{yes}[n \mod 2] = \mathbb{P}\{\sigma \in \mathit{Paths}_{\mathscr{M}}(r.\mathit{term}) \mid \exists i \leq n.\mathsf{M}[\sigma[i]].\mathit{label} = \mathsf{YES} \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad \forall j < i.\mathsf{M}[\sigma[j]].\mathit{label} = \mathsf{UNKNOWN}\}\end{array}$$

$$\begin{array}{l}r'.p_{no}[n \mod 2] = \mathbb{P}\{\sigma \in \mathit{Paths}_{\mathscr{M}}(r.\mathit{term}) \mid \exists i \leq n.\mathsf{M}[\sigma[i]].\mathit{label} = \mathsf{NO} \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad \forall j < i.\mathsf{M}[\sigma[j]].\mathit{label} = \mathsf{UNKNOWN}\}\end{array}$$

Since for each $r' \in \mathsf{M}$, such that $r' \notin A_n$, $r'.p_{yes} = r'.p_{no} = \{0,0\}$, then

$$\begin{array}{l}r.p_{yes}[n+1] = \mathbb{P}\{\sigma \in \mathit{Paths}_{\mathscr{M}}(r.\mathit{term}) \mid \exists i \leq n+1.\mathsf{M}[\sigma[i]].\mathit{label} = \mathsf{YES} \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad \forall j < i.\mathsf{M}[\sigma[j]].\mathit{label} = \mathsf{UNKNOWN}\}\end{array}$$

$$\begin{array}{l}r.p_{no}[n+1] = \mathbb{P}\{\sigma \in \mathit{Paths}_{\mathscr{M}}(r.\mathit{term}) \mid \exists i \leq n+1.\mathsf{M}[\sigma[i]].\mathit{label} = \mathsf{NO} \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad \forall j < i.\mathsf{M}[\sigma[j]].\mathit{label} = \mathsf{UNKNOWN}\}\end{array}$$

which proves the two invariants. □

3.3 Termination and Complexity

The idea of computing both the probability of the set of paths that satisfy a path-formula and the probability of the set of those that do not is motivated by the possibility to exploit an interesting property of *transient* DTMCs [12]. A Markov chain is transient iff all its recurrent states are absorbing. This is the case for the DTMCs that are constructed by the model checking algorithm, in particular the states labelled YES and NO are absorbing and the states labelled UNKNOWN are transient. The probability matrix $\mathbf{P}$ of a generic transient DTMC can be arranged as:

$$\mathbf{P} = \begin{array}{c} \\ E \\ \tilde{E} \end{array} \begin{array}{c} \begin{array}{cc} E & \tilde{E} \end{array} \\ \begin{pmatrix} I & 0 \\ R & Q \end{pmatrix} \end{array} \tag{1}$$

where E and $\tilde{E}$ denote the set of recurrent states and the transient states of the DTMC, respectively. For this kind of Markov chains the following lemma [12, pag. 107] can be easily shown to hold using an inductive argument:

Lemma 3.3 *Let $\mathscr{D} = (S, \bar{s}, \mathbf{P})$ be a transient DTMC, with $\mathbf{P}$ of the form shown in (1), then $\lim_{i\to\infty} Q^i = 0$ and:*

$$\mathbf{P}^i = \begin{array}{c} \\ E \\ \tilde{E} \end{array} \begin{array}{c} \begin{array}{cc} E & \tilde{E} \end{array} \\ \begin{pmatrix} I & 0 \\ (I+Q+\cdots+Q^{i-1})R & Q^i \end{pmatrix} \end{array}$$

Lemma 3.4 *Let s be such that the set $\{s' | \exists k. s' \in \mathscr{R}(s,k)\}$ of states reachable from s is finite; then, for each Φ_1 and Φ_2, CheckUnboundedUntil(s, Φ_1, Φ_2) terminates.*

Proof The statement follows directly from Lemma 3.3 by observing that two transient DTMCs are implicitly considered in function CheckUnboundedUntil. One in which $E = S_{yes}$ while $\tilde{E}$ consists of the set of records labelled UNKNOWN; this DTMC is used to compute the probability mass of the set of paths *satisfying* $\Phi_1 \mathscr{U} \Phi_2$. The other transient DTMC, is the one where E is S_{no} while $\tilde{E}$ is again the set of records labelled UNKNOWN. This second DTMC is used to compute the probability mass of the set of paths that do *not* satisfy $\Phi_1 \mathscr{U} \Phi_2$. Function CheckUnboundedUntil terminates when the sum of the two computed probability values differs from 1.0 by less than a given accuracy bound ε. Note that this difference is in fact the *total remaining probability* in Q^i (where i is the current iteration). Lemma 3.3 guarantees that the threshold is eventually reached.

Lemma 3.4 guarantees that the algorithm always terminates when the set of states reachable from s is finite. For what concerns complexity, the number of iterations that is required to complete the computation depends on the accuracy bound ε and on the *stiffness* of the model. In particular, the number of iterations is bounded by $\frac{\log \varepsilon}{\log(\max_i\{\sum_j Q_{i,j}\})}$.

4 On-the-fly Model checking: a Preliminary Assessment

A prototype of the model checking algorithm has been implemented in Java, together with an interpreter of the PRISM language [13], and used for a first comparison of its performance with that of PRISM[8]. The choice of this specific probabilistic model-checker is justified by the fact that it is the state of the art for what concerns advanced state space reduction techniques.

[8] Details of the experiments are available at `http://j-sam.sourceforge.net`.

Property:	**P1**		**P2**		**P3**		**P4**	
N	O	P	O	P	O	P	O	P
3	1	2	1	2	1	2	1	2
5	2	3	2	3	2	3	2	3
7	3	4	2	3	2	4	3	4
9	8	8	4	4	4	6	4	6

Property:	**P1**		**P2**	
N	O	P	O	P
3	1	3	1	3
5	1	7	1	12
9	1	402	2	711
11	1	6.91s	2	8.027s
15	1	-	2	-
21	2	-	2	-

Table 8: MC time of Self-stabilisation (left) and Dining Philisophers (right)

The first case study we consider is the self-stabilising algorithm of Herman [10, 13]. This algorithm defines a protocol for a network of processes arranged in a ring. Each process can be either *active* or *passive*. A configuration is *stable* when only one process is active. The algorithm guarantees that, when starting from an unstable configuration, the system is able to return to a stable configuration with probability 1 within a finite number of steps. We assume an initial configuration where all the processes are active and four properties that require the full state space to be expanded: (**P1**) the probability to reach a stable configuration within 50 steps is greater than p; (**P2**) the probability to reach eventually a stable configuration is greater than p; (**P3**) the probability to reach a stable configuration within 50 steps while the first process remains active is greater than p; (**P4**) the probability to reach eventually a stable configuration while the first process remains active is greater than p. The model checking time (in milliseconds if not specified otherwise) needed to perform these analyses is reported in Table 8 (left), where N indicates the number of processes in the ring, O the on-the-fly model-checker and P the PRISM model-checker. These first results show that the efficiency of the on-the-fly algorithm is *comparable* with that of PRISM for PCTL[9]. The efficiency of the on-the-fly algorithm increases drastically when, to verify a given property, only a subset of the state space is needed. This is the case for the second case study where a probabilistic variant of the *Dining Philosophers* is considered. We consider two properties: (**P1**) Philosopher 1 is the first to eat within the next 20 steps with probability greater than p; (**P2**) With probability greater than p, philosopher 1 is the first to eat[10]. The execution times of both on-the-fly model checking and PRISM are reported in Table 8 (right). Note that for these specific properties the on-the-fly model-checker is able to give a result within a few milliseconds even for a system composed of 15 or 21 philosophers. PRISM instead raises an *out-of-memory* exception for these cases. Which of the two model checkers is more convenient to use depends on the problem at hand. In highly symmetric cases PRISM is expected to perform better because it can exploit powerful reduction techniques based on the underlying MTBDD structure. Having all techniques available gives the greatest advantage.

5 Conclusions and Future Work

In this paper we have presented an innovative *local, on-the-fly* PCTL model checking approach including both bounded and unbounded modalities. The model checking algorithm is parametric w.r.t. the language and the specific semantic model of interest. The algorithm for unbounded until is new and exploiting a well-known property of transient DTMCs to obtain an efficient procedure to compute the probability of unbounded path formulas with a desired accuracy. Correctness proofs of the algorithms

[9]Experiments have been performed with an Intel Core i7 1.7GHz, RAM 8Gb. For PRISM the model generation time has not been considered.

[10]Properties of single objects are very relevant in e.g. population models [14].

have been provided and a prototype implementation with the PRISM language as front-end has been used to perform a comparison of its efficiency both in memory use and in time and for exact probabilistic model checking. The model checker has also been instantiated and used for fast on-the-fly PCTL model checking for discrete time synchronous population models obtaining a scalability independent of the size of the population [14, 15] Further work is planned on extensions that concern spatial aspects of systems as well as the application to a larger range of case-studies incorporating further probabilistic languages and related semantics. A more detailed comparison with statistical model checking, e.g. [20], or partial order reduction techniques, e.g. [1], is also planned.

References

[1] Christel Baier, Pedro R. D'Argenio & Marcus Grober (2006): *Partial Order Reduction for Probabilistic Branching Time*. *Electr. Notes Theor. Comput. Sci.* 153(2), pp. 97–116, doi:10.1016/j.entcs.2005.10.034.

[2] Christel Baier, Boudewijn R. Haverkort, Holger Hermanns & Joost-Pieter Katoen (2003): *Model-Checking Algorithms for Continuous-Time Markov Chains*. *IEEE Transactions on Software Engineering. IEEE CS* 29(6), pp. 524–541, doi:10.1109/TSE.2003.1205180.

[3] Girish Bhat, Rance Cleaveland & Orna Grumberg (1995): *Efficient On-the-Fly Model Checking for CTL**. In: *LICS*, IEEE Computer Society, pp. 388–397, doi:10.1109/LICS.1995.523273.

[4] Edmund M. Clarke, E. A. Emerson & A. P. Sistla (1986): *Automatic verification of finite-state concurrent systems using temporal logic specifications*. *ACM Trans. Program. Lang. Syst.* 8(2), pp. 244–263, doi:10.1145/5397.5399.

[5] Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper & Mihalis Yannakakis (1992): *Memory-efficient algorithms for the verification of temporal properties*. *Form. Methods Syst. Des.* 1(2-3), pp. 275–288, doi:10.1007/BF00121128.

[6] Alvaro Fernandez-Diaz, Christel Baier, Clara Benac-Earle & Lars-Ake Fredlund (2012): *Static Partial Order Reduction for Probabilistic Concurrent Systems*. *QEST 2012* 0, pp. 104–113, doi:10.1109/QEST.2012.22.

[7] Stefania Gnesi & Franco Mazzanti (2011): *An Abstract, on the Fly Framework for the Verification of Service-Oriented Systems*. In Martin Wirsing & Matthias M. Hölzl, editors: *Results of the SENSORIA Project*, *LNCS* 6582, Springer, pp. 390–407, doi:10.1007/978-3-642-20401-2_18.

[8] Ernst Moritz Hahn, Holger Hermanns, Björn Wachter & Lijun Zhang (2009): *Time-Bounded Model Checking of Infinite-State Continuous-Time Markov Chains*. *Fundam. Inform.* 95(1), pp. 129–155, doi:10.3233/FI-2009-145.

[9] Hans Hansson & Bengt Jonsson (1994): *A Logic for Reasoning about Time and Reliability*. *Formal Aspects of Computing* 6, pp. 512–535, doi:10.1007/BF01211866.

[10] Ted Herman (1990): *Probabilistic Self-Stabilization*. *Inf. Process. Lett.* 35(2), pp. 63–67, doi:10.1016/0020-0190(90)90107-9.

[11] Gerard J. Holzmann (2004): *The SPIN Model Checker - primer and reference manual*. Addison-Wesley.

[12] J. G. Kemeny, J. L. Snell & A. W. Knapp (1976): *Denumerable Markov Chains*. Springer-Verlag, New York, USA, doi:10.1007/978-1-4684-9455-6.

[13] Marta Z. Kwiatkowska, Gethin Norman & David Parker (2004): *Probabilistic symbolic model checking with PRISM: a hybrid approach*. *STTT* 6(2), pp. 128–142, doi:10.1007/s10009-004-0140-2.

[14] Diego Latella, Michele Loreti & Mieke Massink (2013): *On-the-fly Fast Mean-Field Model-Checking*. In Martín Abadi & Alberto Lluch-Lafuente, editors: *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers*, *Lecture Notes in Computer Science* 8358, Springer, pp. 297–314, doi:10.1007/978-3-319-05119-2_17.

[15] Diego Latella, Michele Loreti & Mieke Massink (2013): *On-the-fly PCTL Fast Mean-Field Model-Checking for Self-organising Coordination - Preliminary Version*. Technical Report TR-QC-01-2013, Quanticol Technical Report. Available on-line at `http://www.quanticol.eu`.

[16] Jean-Yves Le Boudec, David McDonald & Jochen Mundinger (2007): *A Generic Mean Field Convergence Result for Systems of Interacting Objects*. In: *QEST07*, IEEE Computer Society Press, pp. 3–18, doi:10.1109/QEST.2007.3. ISBN 978-0-7695-2883-0.

[17] Giuseppe Della Penna, Benedetto Intrigila, Igor Melatti, Enrico Tronci & Marisa Venturini Zilli (2006): *Finite horizon analysis of Markov Chains with the Murphi verifier*. *STTT* 8(4-5), pp. 397–409, doi:10.1007/s10009-005-0216-7.

[18] Amir Pnueli (1977): *The temporal logic of programs*. In: *SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science (SFCS 1977)*, IEEE Computer Society, Washington, DC, USA, pp. 46–57, doi:10.1109/SFCS.1977.32.

[19] Kristin Y. Rozier & Moshe Y. Vardi (2010): *LTL satisfiability checking*. *STTT* 12(2), pp. 123–137, doi:10.1007/s10009-010-0140-3.

[20] Håkan L. S. Younes, Marta Z. Kwiatkowska, Gethin Norman & David Parker (2004): *Numerical vs. Statistical Probabilistic Model Checking: An Empirical Study*. In K. Jensen & A. Podelski, editors: *Proc. 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, *LNCS* 2988, Springer, pp. 46–60, doi:10.1007/978-3-540-24730-2_4.