QUANTICOL

A Quantitative Approach to Management and Design of Collective and Adaptive Behaviours

quantiCOL

http://www.quanticol.eu

TR-QC-09-2014

On-the-fly Probabilistic Model Checking

Extended version

Revision: 0.1; Dec 8, 2014

Author(s): Diego Latella (CNR), Michele Loreti (University of Florence and IMT), Mieke Massink (CNR)

Publication date: Dec 8, 2014

Funding Scheme: Small or medium scale focused research project (STREP)
Topic: ICT-2011 9.10: FET-Proactive 'Fundamentals of Collective Adaptive Systems' (FOCAS)
Project number: 600708

Coordinator: Jane Hillston (UEDIN)
e-mail: Jane.Hillston@ed.ac.uk
Fax: +44 131 651 1426

Part. no.	Participant organisation name	Acronym	Country
1 (Coord.)	University of Edinburgh	UEDIN	UK
2	Consiglio Nazionale delle Ricerche – Istituto di Scienza e Tecnologie della Informazione "A. Faedo"	CNR	Italy
3	Ludwig-Maximilians-Universität München	LMU	Germany
4	Ecole Polytechnique Fédérale de Lausanne	EPFL	Switzerland
5	IMT Lucca	IMT	Italy
6	University of Southampton	SOTON	UK
7	Institut National de Recherche en Informatique et en Automatique	INRIA	France

COOPERATION

Contents

Abstract

Model checking approaches can be divided into two broad categories: global approaches that determine the set of all states in a model $\mathcal{M}$ that satisfy a temporal logic formula Φ, and local approaches in which, given a state s in $\mathcal{M}$, 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 explored on a number of benchmark applications in comparison with the global PCTL model checker PRISM.

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 $\mathcal{M}$ that satisfy a temporal logic formula Φ, and *local* approaches in which, given a state s in $\mathcal{M}$, the procedure determines whether s satisfies Φ [1, 2]. 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 [3] 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) [4], a probabilistic extension of the temporal logic *Computation Tree Logic* (CTL) [3] that includes both the bounded and unbounded until operator. To the best of our knowledge the only algorithm for on-the-fly model checking for probabilistic processes is the one proposed in [5] that only considers the fragment of the PCTL without unbounded until. On the contrary, the local model-checking algorithm considered in this paper considers full PCTL. This is an important point. Indeed, the use of full PCTL is incompatible with the application of specific techniques, like for instance statistical model checking [6], that can be used only for properties with a bounded temporal horizon. A further innovative aspect is that 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 [7] 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 such a semantics [8] described in more detail in recent work by the authors [7, 9]. The mean-field approximation has proved a successful, and highly scalable, 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 [7, 9]) 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. those 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 [10] which in turn is used to compare the efficiency of the on-the-fly algorithm with that of PRISM for a number of benchmark case studies. This paper extends the work described in [11] in particular for what concerns the more detailed explanation of the algorithm and the study of the efficiency of the approach.

Related work. In the context of qualitative model checking of temporal logics such as CTL [3], LTL [12, 13] and CTL*[2], local model checking algorithms have been proposed to mitigate the state space explosion problem using an on-the-fly approach [1, 2, 14, 15]. They 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 some cases.

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 [16, 17]. A clear advantage of these global algorithms is that results are obtained for *all* states of the model at once, 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. Moreover, the model-checking procedures can be reduced to combinations of existing well-known and optimised algorithms for Markov chains such as transient analysis [16]. In the context of Markov Decision Processes (MDP) partial order reduction techniques have been explored to obtain

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

[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, under the condition that the population is sufficiently large.

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

Table 1: Satisfaction relation for PCTL.

state space reduction [18]. 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 [19]. As already mentioned, to the best of our knowledge, the only other algorithm for on-the-fly probabilistic model checking is the one proposed in [5], which addresses only the bounded fragment of PCTL, whereas we are providing also an efficient algorithm for unbounded until in the current paper.

2 Probabilistic Computation Tree Logic

In this section we briefly recall the definition of the *Probabilistic Computation Tree Logic* (PCTL) [4], a probabilistic extension of the temporal logic CTL [3], for the expression of properties of Discrete Time Markov Chains (DTMCs) and Markov Decision Processes (MDPs). The syntax of PCTL is the following:

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

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 $\mathcal{P}_{\bowtie p}(\varphi)$. Informally, a state s in a DTMC satisfies $\mathcal{P}_{\bowtie p}(\varphi)$ 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 $\langle \mathcal{M}, \ell \rangle$ where $\mathcal{M}$ is a DTMC with state set $\mathcal{S}$ and $\ell : \mathcal{S} \to 2^{\mathscr{P}}$ associates each state with a set of atomic propositions; for each state $s \in \mathcal{S}$, $\ell(s)$ is the set of atomic propositions true in s. In the following, we assume $\mathbf{P}$ be the one step probability matrix for $\mathcal{M}$; we abbreviate $\langle \mathcal{M}, \ell \rangle$ with $\mathcal{M}$, when no confusion can arise. A path σ over $\mathcal{M}$ 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 $Paths_{\mathcal{M}}(s)$ denote the set of all infinite paths over $\mathcal{M}$ 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 $\mathcal{M}$ and the logic are formally defined in Table 1. For every path formula φ, the set $\{\sigma \in Paths_{\mathcal{M}}(s) | \sigma \models \varphi\}$ is a *measurable set* [17].

3 On-the-fly Probabilistic Model Checking

We introduce a local on-the-fly model checking algorithm for PCTL on labelled DTMC $\langle \mathcal{M}, \ell \rangle$. 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 incrementally. For each generated state 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 when possible. However, the proposed model checking algorithm is not only based on graph generation. While the relevant part of the state space is generated, the satisfaction probabilities of path formulas are also computed and updated in an on-the-fly fashion.

```
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

The proposed algorithm abstracts from any specific modelling 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.

Let us first describe the main algorithm in more detail. Let proc be the (generic) type of *probabilistic process terms* and let formula and path_formula be the types of *state-* and *path-* PCTL formulas, respectively. Moreover, let lab denote the type of *atomic propositions*. The abstract interpreter can then be modelled by means of two functions: next and lab_eval. Function next associates a list of pairs (proc, float) to each element of type proc. This list gives the terms, i.e. states, that can be reached in *one* step from the given state s 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 the functions 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, Check returns the value of lab_eval. When given a non-atomic state-formula, Check calls itself recursively on sub-formulas, in case the latter 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[3].

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 $\varphi = \mathcal{X}\,\Phi$ 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 $\Phi_1\,\mathcal{U}^{\leq k}\,\Phi_2$ or $\Phi_1\,\mathcal{U}\,\Phi_2$ 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 $\mathcal{M}$ the complete discrete time stochastic process associated with s by the formal semantics of the language. The correctness of the algorithm is formalised by the following theorem:

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

Proof. The theorem is proven by induction on the structure of Φ. The more involving parts concern the proof of the theorem for the path formulas concerning bounded and unbounded until. These are provided as Lemma 3.2, Lemma 3.4 and Lemma 3.5 together with an outline of their proofs in the following sections.

□

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

```
1  float CheckPath( s : proc , φ : path_formula ) {
2   switch φ {
3    case X Φ: {
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

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 $\Phi_1 \mathcal{U}^{\leq k} \Phi_2$. This function takes as parameters a state s, state formulas Φ_1 and Φ_2, and the bound k. As we will see in the following, we propose an iterative solution to compute the probability of the set of paths satisfying $\Phi_1 \mathcal{U}^{\leq k} \Phi_2$. This procedure differs from the algorithm proposed in [5], where a recursive algorithm is proposed, and from the standard global PCTL model checking approach where the whole state space is considered to compute the requested probability value. From the latter algorithm we do adopt a state labelling strategy dividing generated states into three kinds: YES, NO and UNKNOWN as will be explained later on.

To compute $\mathbb{P}\{\sigma \in Paths_{\mathcal{M}}(s) \mid \sigma \models_{\mathcal{M}} \Phi_1 \mathcal{U}^{\leq k} \Phi_2\}$ the algorithm is split into two phases. First, the function CheckBoundedUntil 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[4]. The use of data structure M enables *memoization* [21] facilitating the reuse of the probability values computed earlier in different sub-formulae. Structure M is a *hashmap*[5] 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 (BURecord, float). Intuitively, given a BURecord r, a pair (r′, 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 $\mathbb{P}\{\sigma \in Paths_{\mathcal{M}}(s') \mid \sigma \models_{\mathcal{M}} \Phi_1 \mathcal{U}^{\leq i} \Phi_2\}$[6].
- label: a label taking a value in {YES, NO, UNKNOWN}. This field takes value YES when term s' satisfies Φ_2. When term s' satisfies neither Φ_2 nor Φ_1 the field label takes value NO and when it satisfies only Φ_1 it takes value UNKNOWN.

[4] A similar approach is used in [20] to analyse infinite Markov chains. However in [20] 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.

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

[6] 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.

```
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

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 $(r, p) \in r'$.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 of Table 4), 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 of Table 4). For each

```
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

state s to be expanded, the list of states s' reachable in one step from s 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 checks 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, $\mathbb{P}\{\sigma \in Paths_{\mathcal{M}}(s) \mid \sigma \models_{\mathcal{M}} \Phi_1 \mathcal{U}^{\leq k} \Phi_2\}$ is 0.0. If $S_{yes} \neq \emptyset$, function CheckBoundedUntil enters the second phase, which is 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. An small illustration of the computation phase is given in Fig. 1 and Fig. 2 showing the probabilities p[i] to reach a YES-labelled state in at most i steps. Note that the label of the states in these figures only show the probability p[i] and not the complete float array p.

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.

3.2 Correctness of the Bounded Until Algorithm

The following lemma shows the correctness of the algorithm for the bounded until operator.

Lemma 3.2 *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. M[s].*label* = YES *and* $p = 1.0$;
2. M[s].*label* = NO *and* $p = 0.0$;
3. M[s].*label* = UNKNOWN, $S_{yes} = \emptyset$ *and* $p = 0.0$;

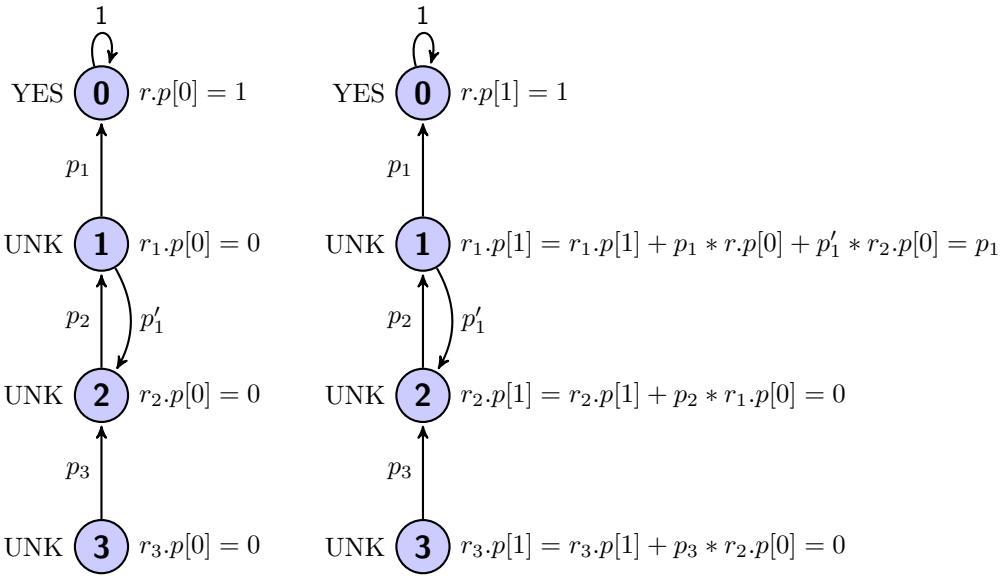

Figure 1: Probability to reach YES in zero steps (left) and in at most one step (right)

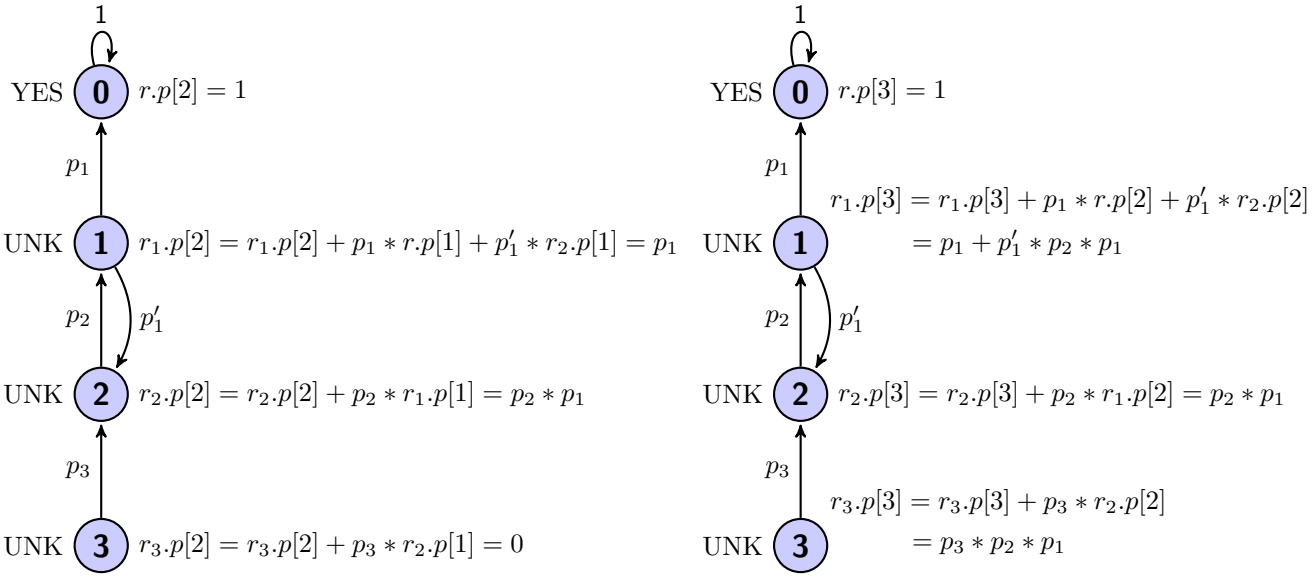

Figure 2: Probability to reach YES in at most two steps (left) and in at most three steps (right)

4. $\mathsf{M}[s].label = \mathsf{UNKNOWN}$, $S_{yes} \neq \emptyset$ *and*

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

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:

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

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

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

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 $A = S_{yes} \subset \mathcal{R}(s,k)$.
Induction Hypothesis: For each $i \leq n$ we have that at line 32 the following hold:

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

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

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

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

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

which proves the correctness of the two invariants.

□

3.3 Computing Unbounded Until Probability

In this section we present function CheckUnboundedUntil that is used to compute the probability of the set of paths satisfying $\Phi_1 \mathcal{U} \Phi_2$ starting from a state s. But before presenting the details of the proposed algorithm, we outline the main ideas on which this new algorithm is based. Similarly to function CheckBoundedUntil considered in the previous section, function CheckUnboundedUntil is structured in two phases: an *expansion phase* and a *computation phase*. At the end of the expansion phase, just before starting the actual computation phase, the DTMC that is generated is of a particular form (by construction), namely all the states labelled YES or NO are absorbing, and all other states, labelled UNKNOWN, are transient, i.e. there is a non-zero probability to never return to such a state. This particular kind of DTMCs are also known as *transient* DTMCs [22]. More formally, a Markov chain is transient iff all its recurrent states are absorbing. Transient DTMCs have an interesting property, namely that in the long run the probability mass in the transient states tend to zero, and all the probability mass accumulates in the absorbing states. We exploit this property in the computation phase to reach a predefined level of accuracy for the computed probability. To do so, we need to compute for each state both the probability mass of the set of paths *satisfying* $\Phi_1 \mathcal{U} \Phi_2$ and that of the set of paths *not* satisfying $\Phi_1 \mathcal{U} \Phi_2$. When, for each state, the sum of the probabilities of both sets is close to 1, upto an accuracy ε, this indicates that enough iterations have been performed such that the probability has been computed with the desired level of accuracy. A closer inspection of the algorithm for the bounded until operator shows that both probabilities can be computed in an iterative way, and that to do so the probability in the next iteration uses only that computed in the current iteration. So, only two values for the probabilities need to be kept in memory[7].

Let us first recall more formally the relevant property of transient DTMCs. The probability matrix $\mathbf{P}$ of a generic transient DTMC can be arranged in four sub-matrices as follows:

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

where E and $\tilde{E}$ denote the set of recurrent states and the transient states of the DTMC, respectively, I denotes the identity matrix, 0 the zero matrix, R the matrix of transitions going from transient states to recurrent states and Q the matrix of transitions from transient to transient states. For this kind of Markov chains the following lemma [22, pag. 107] can be easily shown to hold using an inductive argument:

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

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

In the algorithm we will implicitly use this property twice, once with set E the set of states labelled by YES, and once with set E the set of states labelled by NO (and $\tilde{E}$ the set of states labelled by UNKNOWN in both cases). Lemma 3.3 plays an important role in the proof of termination of the algorithm. A further important aspect of the algorithm is that not all states generated during the expansion phase will be involved in the computation phase. This holds in particular for states that form a bottom strongly connected component and that are all labelled by NO or all labelled by YES. These strongly connected components do not need to be detected by a separate routine, but will be automatically excluded due to the right choice of an initial set of active states A that will consist of

[7] The reason for not doing so in the algorithm for bounded until is that these probability values are also useful in case one needs to verify a series of bounded until path formulae in which the time bound takes a series of consecutive values.

```
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

only those states that are labelled YES or NO, and that moreover have an incoming transition that originates in a state that is *not* labelled by YES or NO. The set of active states will be updated in every iteration in the computations phase, but this will never lead to the inclusion of further states on a bottom strongly connected component. This will be explained in more detail later on. To check the label of incoming transitions to a state is facilitated by the fact that during the expansion phase the list of incoming transitions is memorised on the fly and easily accessible for later use. We now proceed by describing the algorithm in more detail.

As mentioned earlier, function CheckUnboundedUntil, defined in Table 8, 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 $\mathbb{P}\{\sigma \in Paths_{\mathcal{M}}(s) \mid \sigma \models_{\mathcal{M}} \Phi_1 \,\mathcal{U}\, \Phi_2\}$, 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 is to contain the probabilities for a state to satisfy (p_{yes}) and not to satisfy (p_{no}) the unbounded until path formula $\Phi_1 \,\mathcal{U}\, \Phi_2$. 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 8, 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 state is found. Before starting the *computation phase*, it is first checked whether S_{yes} is empty (Line 31 in Table 8) . 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 labelled UNKNOWN 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 remaining records that *cannot reach* NO-labelled records are added to S_{yes}, labelled by YES and their probability value 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 of such BSCCs cannot reach states that are labelled YES (or NO), but they can be treated as states labelled NO. An example of such a transformation is shown in Fig. 3 for a simple DTMC.

The *computation phase* (starting at line 39 in Table 8) operates on a set of *active* records A. Initially, A is $\{r \in S_{yes} \cup S_{no} \mid \exists r' \notin S_{yes} \cup S_{no} : r \prec r'\}$, i.e. only those YES or NO-labelled states

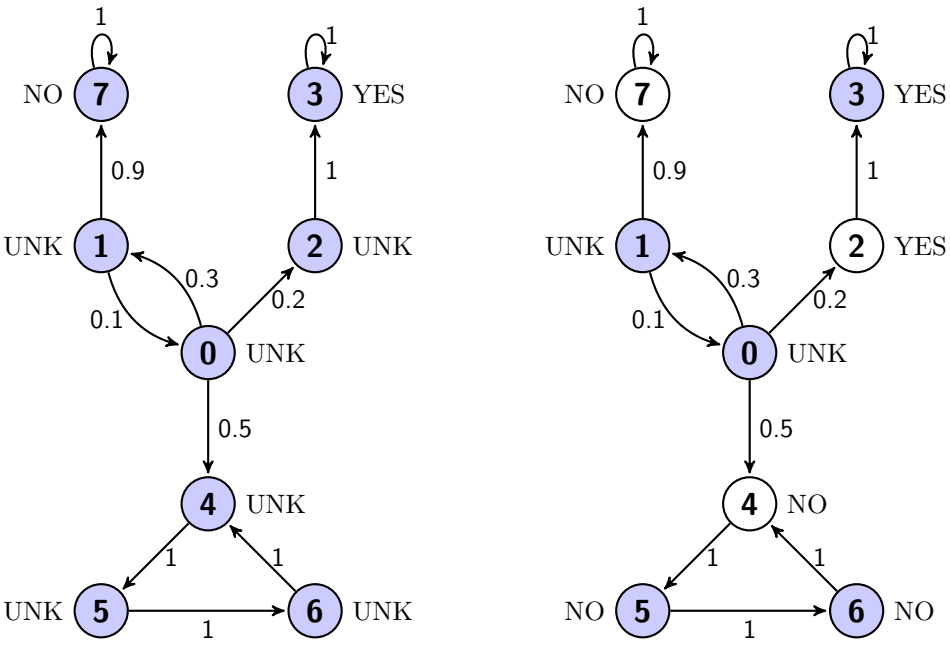

Figure 3: Transformation of the generated model (left) at the end of the expansion phase into the model at the right at the start of the computation phase. Note the label change of states 2, 4, 5 and 6.

that have an incoming transition from a state that is not labelled by YES or NO. In the example shown in Fig. 3 the initial set A is shown in the figure on the right by the white nodes.

At the end of each iteration (line 42 in Table 8) 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, $\mathsf{r.p_{yes}}[i \bmod 2]$ contains the probability mass of the set of paths starting from $\mathsf{r.term}$ 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, $\mathsf{r.p_{no}}[i \bmod 2]$ stores the probability of the set of paths starting from $\mathsf{r.term}$ 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: $\mathsf{r.p_{yes}}[i \bmod 2] + \mathsf{r.p_{no}}[i \bmod 2] \geq 1 - \varepsilon$, where ε is a given accuracy level. The result for a few iterations of the computation for the example in Fig. 3 is shown in Table 7. The table shows the nodes and their labels in the first two rows, followed by the indication of the initial set of active nodes in set A. It then shows the initial values of the probabilities in p_{yes} and p_{no} and their computed values in three consecutive iterations. After each iteration also the sum of p_{yes} and p_{no} is given. This sum is getting closer to 1 in every iteration, and after the last iteration ($i = 2$) this value differs less than $\epsilon = 0.02$ from 1, determining the termination of the algorithm. The values for p_{yes} and p_{no} in each iteration are computed in a similar way as in the algorithm for bounded until, except that now only the current and next values are kept in each iteration.

3.4 Correctness, Termination and Complexity of the Unbounded Until Algorithm

The following lemma states the partial correctness of the algorithm for the unbounded until operator.

Lemma 3.4 *For each s, Φ_1 and Φ_2, let $\mathsf{CheckUnboundedUntil}(s, \Phi_1, \Phi_2) = p$ and M be the data structure obtained at the end of the expansion phase, one of the following holds:*

1. $\mathsf{M}[s].label = \mathsf{YES}$ *and* $p = 1.0$;

node	0	1	2	3	4	5	6	7
label	U	U	Y	Y	N	N	N	N
A(ctive nodes)			x		x			x
p_{yes}	0, 0	0,0	1,1	1,1	0,0	0,0	0,0	0,0
p_{no}	0, 0	0,0	0,0	0,0	1,1	1,1	1,1	1,1
$i = 0, p_{yes}(i \ mod\ 2, (i+1)\ mod\ 2)$	0, 0.2	0, 0	1, 1	1, 1	0,0	0, 0	0,0	0,0
$i = 0, p_{no}(i \ mod\ 2, (i+1)\ mod\ 2)$	0, 0.5	0, 0.9	0,0	0, 0	1,1	1,1	1,1	1,1
$p_{yes} + p_{no}$	0, 0.7	0, 0.9	1,1	1, 1	1,1	1, 1	1,1	1,1
A(ctive nodes)	x	x	x		x			x
$i = 1, p_{yes}(i \ mod\ 2, (i+1)\ mod\ 2)$	0.2, 0.2	0.02, 0	1, 1	1, 1	0,0	0, 0	0,0	0,0
$i = 1, p_{no}(i \ mod\ 2, (i+1)\ mod\ 2)$	0.77, 0.5	0.95, 0.9	0,0	0, 0	1,1	1, 1	1, 1	1,1
$p_{yes} + p_{no}$	0.97, 0.7	0.97, 0.9	1,1	1, 1	1,1	1,1	1,1	1,1
A(ctive nodes)	x	x	x		x			x
$i = 2, p_{yes}(i \ mod\ 2, (i+1)\ mod\ 2)$	0.2, 0.2	0.02, 0.02	1, 1	1, 1	0,0	0, 0	0, 0	0,0
$i = 2, p_{no}(i \ mod\ 2, (i+1)\ mod\ 2)$	0.77, 0.785	0.95, 0.977	0,0	0, 0	1,1	1, 1	1, 1	1,1
$p_{yes} + p_{no}$	0.97, 0.985	0.97, 0.997	1,1	1, 1	1,1	1,1	1,1	1,1

Table 7: Three iterations of the computation of p_{yes} and p_{no} probabilities for the example of Fig. 3. For $\varepsilon = 0.02$ after iteration $i = 2$ the threshold of $p_{yes} + p_{no} \geq 1 - \varepsilon$ is satisfied and the algorithm terminates.

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. $\mathsf{M}[s].label = \mathsf{UNKNOWN}$*,* $S_{no} = \emptyset$ *and* $p = 1.0$*;*

5. $\mathsf{M}[s].label = \mathsf{UNKNOWN}$*,* $S_{yes}, S_{no} \neq \emptyset$ *and*

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

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:

$A = \{r' | \exists r'' \in S_{yes} \cup S_{no} : r' \preceq^i r''\}$
$\forall s'.\mathsf{M}[s'] = r \neq \bot,$
$r.p_{yes}[i \mod 2] = \mathbb{P}\{\sigma \in Paths_{\mathcal{M}}(r.term) \mid \exists i' \leq i.\mathsf{M}[\sigma[i']].label = \mathsf{YES} \wedge \forall j' < i'.\mathsf{M}[\sigma[j']].label = \mathsf{UNKNOWN}\}$

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

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:

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

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

```
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 = {r ∈ S_yes ∪ S_no | ∃r′ ∉ S_yes ∪ S_no : r ≺ r′}
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 8: Function CheckUnboundedUntil

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 3.4, 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 3.2 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:

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

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

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 8):

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

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

By induction hypothesis, we have that:

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

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

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

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

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

which proves the two invariants. □

Termination of the algorithm for unbounded until is formalised by the following lemma. Its proof is a direct consequence of the property of transient DTMC stated in Lemma 3.3.

Lemma 3.5 *Let s be such that the set $\{s' \mid \exists k.s' \in \mathcal{R}(s,k)\}$ of states reachable from s is finite; then, for each Φ_1 and Φ_2, $\mathsf{CheckUnboundedUntil}(s, \Phi_1, \Phi_2)$ 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* $\Phi_1 \mathcal{U} \Phi_2$. 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 $\Phi_1 \mathcal{U} \Phi_2$. 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 3.5 guarantees that the algorithm always terminates when the set of states reachable from s is finite.

For what concerns the computational complexity of the algorithm, 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

$$\frac{\log \varepsilon}{\log(\max_i\{\sum_j Q_{i,j}\})}$$

As an example, consider the model in Fig. 4 with probability δ chosen very small to obtain a stiff model in which the probability mass flows very slowly to the absorbing nodes labelled YES and NO. If the accuracy bound is chosen in the order of 10^{-6}, the number of iterations needed to complete the computational phase inversely increases with respect to the magnitude of δ. If δ is 0.1, the computation terminates after 118 iterations. If δ is 10^{-6} the iterations needed to compute the expected probability are $12,280,474$. In the table below we report the number of iterations needed to complete the computation when δ varies from 10^{-1} to 10^{-6}.

δ	*iterations*
10^{-1}	$\sim 1.2 \cdot 10^2$
10^{-2}	$\sim 1.2 \cdot 10^3$
10^{-3}	$\sim 1.2 \cdot 10^4$
10^{-4}	$\sim 1.2 \cdot 10^5$
10^{-5}	$\sim 1.2 \cdot 10^6$
10^{-6}	$\sim 1.2 \cdot 10^7$

In all the considered cases the on-the-fly algorithm is able to compute the correct probability value without any significative deviation from the expected result.

In case of PRISM some care needs to be taken. To analyse the model, two different standard iterative numeric methods are available, Jacobi and Gauss-Seidel. For both methods the threshold epsilon and the termination criterion must be provided. If the relative termination criterium is chosen to establish when to stop iterating, the relative difference between the results of two consecutive iterations is used. When the latter is smaller than the threshold epsilon it is assumed that the final value has been reached. However, in stiff models this may lead to a premature end of the iterations, leading to a sometimes significant underestimation of the probability. As an illustration, for the model with $\delta = 10^{-6}$ and termination epsilon also 10^{-6} PRISM gives probability 0.600 to eventually end up in state $s = 5$ and probability 0.0999 to eventually end up in state $s = 6$. Clearly, those probabilities do not sum to 1, as one would have expected in this particular case. To get a result closer to the actual probabilities, the threshold epsilon must be chosen much smaller (we tried 10^{-9} giving probability 0.853485111 to eventually end up in state $s = 5$). This however also leads to a much higher number of required iterations $(23,386,049)$ performed in about 6 seconds. This is still considerably more than the number of iterations required with the on-the-fly approach where "only" $12,480,272$ iterations, performed in about 2 seconds, are needed to compute the actual probabilities 0.888888 to reach $s = 5$ and 0.111111 to reach $s = 6$.

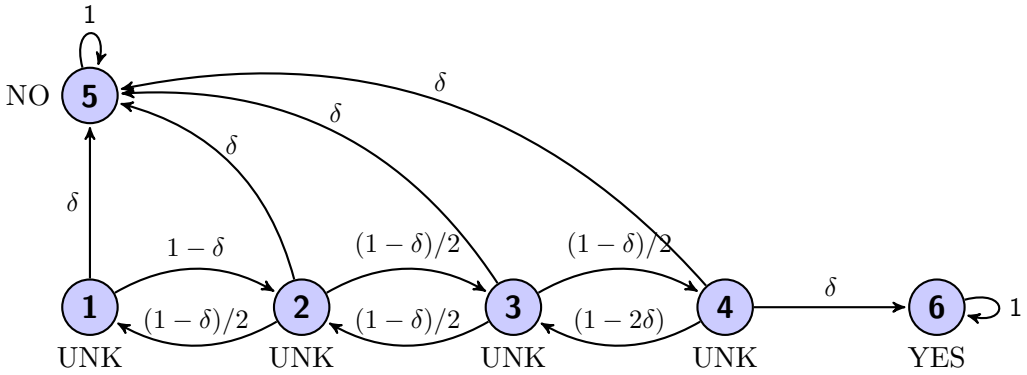

Figure 4: Example of a stiff model for small values of δ.

4 On-the-fly Model Checking at Work

Although it is commonly known that on-the-fly techniques give an advantage when the model and the property of interest are such that only a part of the state space needs to be searched to reach a result, we are also interested in whether this way really larger models can be analysed and whether the performance in case of a full state space search is still acceptable.

In this section, we present some experimental results obtained by using a prototype Java implementation of the on-the-fly model checking algorithm proposed in the previous sections to perform analysis of a set of significative case studies. The purpose of these experiments is to get a more detailed insight in the performance of the on-the-fly probabilistic model checker, both in terms of generated state space and evaluation time. Since the proposed implementation is able to take as input PRISM [17, 10] specifications, we use a selection of benchmark models for this comparison as provided by the PRISM benchmark suite [23] extended with the SEIR computer epidemic model [24] and the Randomised Dining Philosophers [25].

In order to have a base for comparison, we conduct the same analyses with the PRISM model checker[8]. The choice of this specific probabilistic model-checker is justified by the fact that it is one of the state of the art probabilistic model-checkers for what concerns advanced state space reduction techniques.

4.1 PRISM

PRISM [17, 10] stands for Probabilistic Symbolic Model Checker. The core model-checking algorithms in PRISM are mainly implemented in C++, whereas parts such as the user interface and the parsers are written in Java. A modified version of the CUDD package is used for an efficient state space representation. The strength of PRISM lays in the generation of multi-terminal binary decision diagrams (MTBDD) to represent transition matrixes. Four types of model checking engines can be used in PRISM: *MTBDD*, *sparse*, *hybrid* and *explicit*. In the following we use PRISM_M, PRISM_S, PRISM_H and PRISM_E to refer to the four engines, respectively.

PRISM_M, PRISM_S, PRISM_H construct the model in a *symbolic* fashion and rely on specific data structures such as binary decision diagrams (BDDs) and multi-terminal BDDs (MTBDDs). Numerical computation performed during model checking, however, is carried out differently for the three engines: *MTBDD* engine is purely based on MTBDDs and BDDs; the *sparse* engine uses sparse matrices; the *hybrid* engine combines the other two. PRISM_E performs all aspects of model construction and model checking using explicit-state data structures. Models are typically stored as sparse matrices or variants thereof. This engine is implemented purely in Java.

[8]Details of the experiments are available at `http://j-sam.sourceforge.net/otfpmc/`. The version of PRISM used is 4.2.beta1, the 64 bit version for Mac OS X.

4.2 Description of experiments.

For each of the considered case studies we identified a set of representative PCTL formulas that are verified by using the on-the-fly probabilistic model checker with an accuracy of 10^{-6}. For each execution we collect the *total model checking time* and *the fraction of the state space* used in the computation. The same experiments are also performed by using the various model checking engines provided by PRISM. For each of these we report the *model generation time*, the *model checking time* and the *generated state space*. Experiments have been performed with an Intel Core i7 $1.7GHz$, RAM $8Gb$ running *Mac OS X 10.10* and using PRISM version 4.2.beta1, 64 bit.

4.3 Case Studies

In this section we use four case studies. Two of these case studies, the *Bounded Retransmission Protocol* and *Herman's self-stabilisation protocol*, are borrowed from the PRISM benchmark suite [23]. The other two examples, the *SEIR computer epidemic model* and the *Randomised Dining Philosophers*, are used to highlight some specific characteristics of the on-the-fly model checker.

4.3.1 Bounded Retransmission Protocol (BRP)

The Bounded Retransmission protocol (BRP) [26] is a variant of the alternating bit protocol. It sends a file divided into a number of chunks, but each chunk can be retransmitted only a bounded number of times. The number of chunks and the maximal number of retransmissions are parameters of the model. We consider the version provided by the PRISM benchmark suite [23] for varying values of the parameters giving the number of chunks of a file (N) and the maximum number of retransmissions (MAX), ranging from 16 to 256 and from 2 to 5, respectively. In Table 9 we report the number of states and transitions of the considered instances of BRP.

We consider the following properties: (**P1**) the probability that a file is successfully transmitted; (**P2**) the probability that a file is successfully transmitted while the number of retransmissions per chunk remain below bound 2. These can be stated formally as:

(P1) $\mathcal{P}_{=?}(true\ \mathcal{U}\ \ s = 4 \wedge i = N)$

(P2) $\mathcal{P}_{=?}(nrtr < 2\ \mathcal{U}\ \ s = 4 \wedge i = N)$

The first formula, **(P1)**, represents an *unconditional unbounded reachability* property. In this case we are only interested in identifying the probability to eventually reach a given state indicating the successful transmission of a file (the state satisfying $s = 4 \wedge i = N$) . The second formula, **(P2)**, represents a *conditional unbounded reachability* property. In this case we are interested in studying the probability to eventually reach a state indicating the successful transmission of a file (the one satisfying $s = 4 \wedge i = N$) but under the assumption that all the traversed states satisfy some conditions, in this case the constraint that the number of retransmissions for each chunk is always less than two (i.e. $nrtr < 2$).

Experimentation results The execution times for the unconditional unbounded reachability property **(P1)** are reported in Fig. 5. Each subfigure shows the execution times for the on-the-fly model checker and for the four different PRISM engines. The times are reported in seconds and there is one subfigure for each value of the number of chunks N considered. The execution times for **(P1)** provides insight in the performance of the on-the-fly model checker when the whole state space needs to be considered to verify the property. So this is a situation in which no particular advantage is obtained by using an on-the-fly approach with respect to a global model checking approach. The BRP model has no particular symmetry that can be exploited by the MTBDD approach. Both aspects are reflected in the execution times shown in Fig. 5.

N	MAX	**States**	**Transitions**
16	2	677	867
16	3	886	1155
16	4	1095	1443
16	5	1304	1731
32	2	1349	1731
32	3	1766	2307
32	4	2183	2883
32	5	2600	3459
64	2	2693	3459
64	3	3526	4611
64	4	4359	5763
64	5	5192	6915
64	2	146013	198003
64	3	292013	396003
64	4	438013	594003
64	5	438013	594003
128	2	5381	6915
128	3	7046	9219
128	4	8711	11523
128	5	10376	13827
256	2	5381	6915
256	3	7046	9219
256	4	8711	11523
256	5	10376	13827

Table 9: BRP: size of considered models.

One can observe that the proposed on-the-fly algorithm is in general faster than any of the four PRISM engines and for both properties (see Figure 6 for execution times for property **(P2)**). As can be expected, this holds in particular for the conditional reachability property where, thanks to the fact that only a fraction of the state space is generated, the on-the-fly algorithm is one order of magnitude faster than the PRISM$_M$ and the explicit state engines. That the on-the-fly approach is also faster in case of the unconditional reachability property **(P1)** for this particular model shows that the on-the-fly approach may provide comparable execution times also for pure reachability properties in which all states need to be considered. As shown later, this does not hold for all possible models, since models in which strong symmetry aspects can be exploited are analysed much faster with PRISM.

For the sparse and the hybrid engines the performance in time is comparable to that of the on-the-fly approach, though for higher values of N the sparse and hybrid engines seem to be a little faster. Note, however, that the execution times of the PRISM engines shown in Fig. 5 and Fig. 6 only include the actual model checking times and *not* the time for model construction. The latter are shown in Fig. 7. In the on-the-fly approach both times are included in the result.

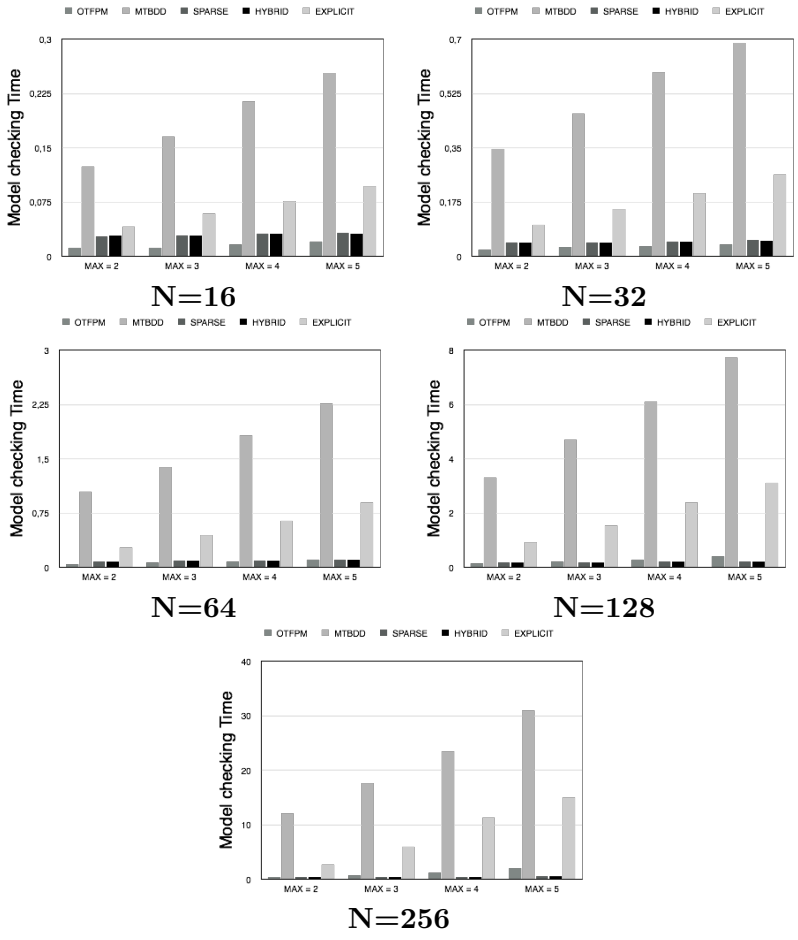

Figure 5: BRP: execution times of model checking of **P1** (in seconds)

4.3.2 Herman's self-stabilisation protocol (HSS).

The self-stabilising algorithm of Herman [27, 17] defines a protocol for a network of an odd number of processes arranged in a ring. Each process can be either *active* or *passive*. A configuration is defined *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

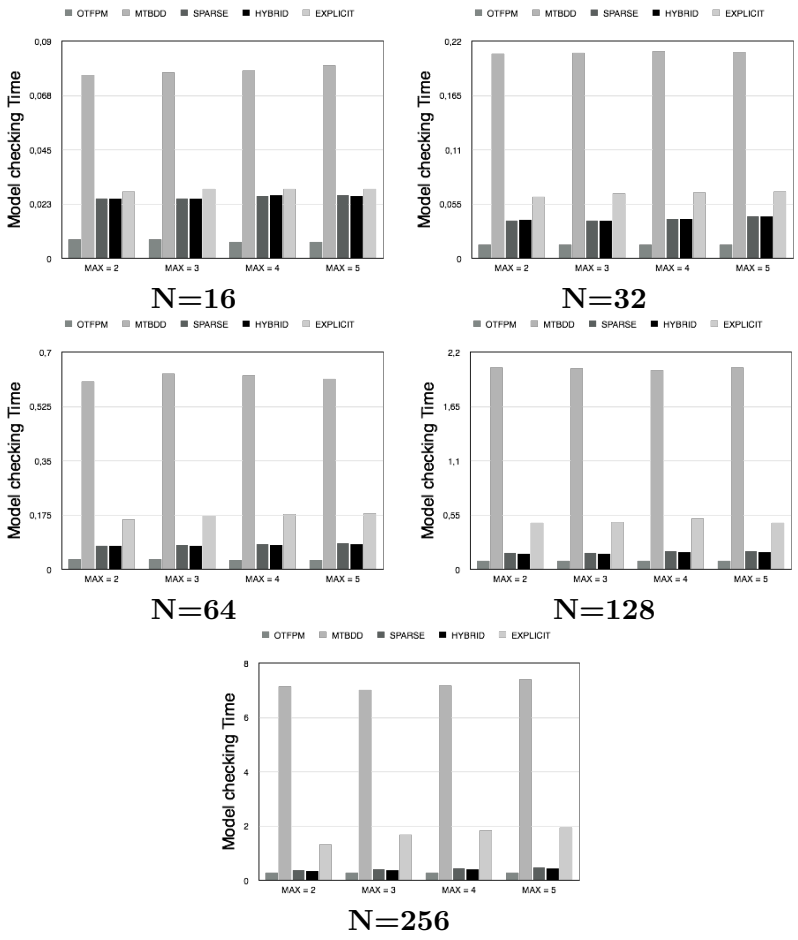

Figure 6: BRP: execution times of model checking of **P2** (in seconds)

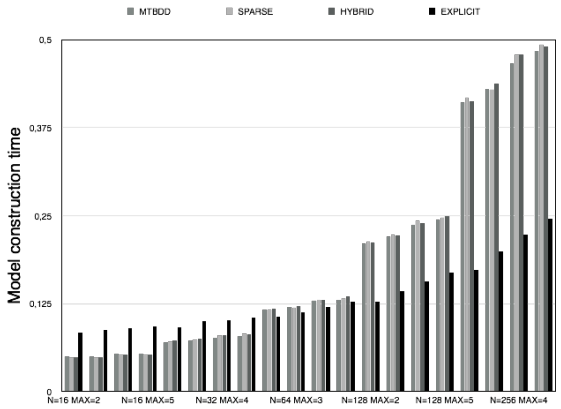

Figure 7: BRP: PRISM model construction time.

Processes	**States**	**Transitions**
3	8	28
5	32	244
7	128	2188
9	512	19684
11	2048	177148
13	8192	1594324
15	32768	14348908

Table 10: HSS: size of considered models.

number of steps. We use the version of the PRISM benchmark suite [23] for a network consisting of 3 up to 15 processes. In Table 10 we report the number of states and transitions of the considered instances of HSS.

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. We use the following formalisation in PCTL:

(P1) $\mathcal{P}_{>p}(\text{tt } \mathcal{U}^{\leq 50} \text{ stable})$

(P2) $\mathcal{P}_{>p}(\text{tt } \mathcal{U} \text{ stable})$

(P3) $\mathcal{P}_{>p}(\text{proc1active } \mathcal{U}^{\leq 50} \text{ stable})$

(P4) $\mathcal{P}_{>p}(\text{proc1active } \mathcal{U} \text{ stable})$

Similarly to the BRP example considered in the previous section, also in this case we have an *unbounded reachability* property (**P2**) and a *conditional unbounded reachability* (**P4**). However, for the example considered in this section, we consider also *bounded reachability* and *bounded conditional reachability*, **P1** and **P3**, respectively, imposing a limit on the number of steps needed to reach a stable state.

Experimentation results The collected execution times for the HSS model are reported in Figure 9 and in Table 11. Figure 9 shows four subfigures, one for each of the four properties. Each subfigure shows the execution times for the on-the-fly model checker and the four PRISM engines for 3, 5, 7 and 9 processes in the token ring, respectively. In Table 11 the execution times for larger values of N are shown, in particular for $N = 11$ and $N = 15$.

Though for a small number of processes the execution times of the on-the-fly model checker are mostly comparable to those of the PRISM engines, for higher numbers of processes the execution time of the on-the-fly approach is growing exponentially while the PRISM engines are performing much better. The latter is due to the strong symmetry of the HSS model that the symbolic engines can exploit to significantly reduce the state space. This is also witnessed by the little time needed by the PRISM symbolic engines to generate the DTMC (see Figure 8). In the on-the-fly case the symmetry of the model does not provide any particular advantage. The HSS model is in fact *dense* in the sense that all the states in the generated DTMC are already visited after just one step of the computation. Since the number of states/transitions in the generated DTMC increase exponentially with the number of processes considered in the model, the impact on the on-the-fly algorithm (whose complexity is polynomial in the number of states/transitions) is more significative for higher values of N (see Table 11). Note that this aspect is evident also in the PRISM$_E$ engine where, for $N = 15$, an *out of memory* exception arises.

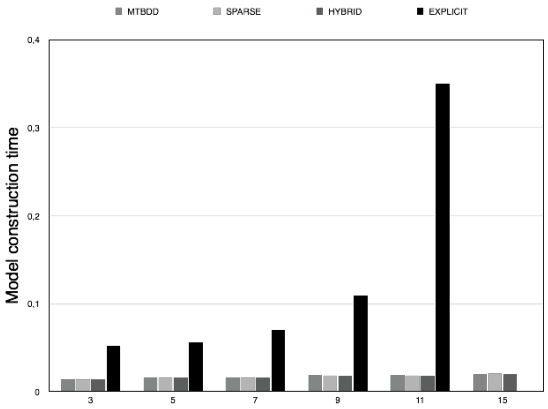

Figure 8: BRP: PRISM model construction time.

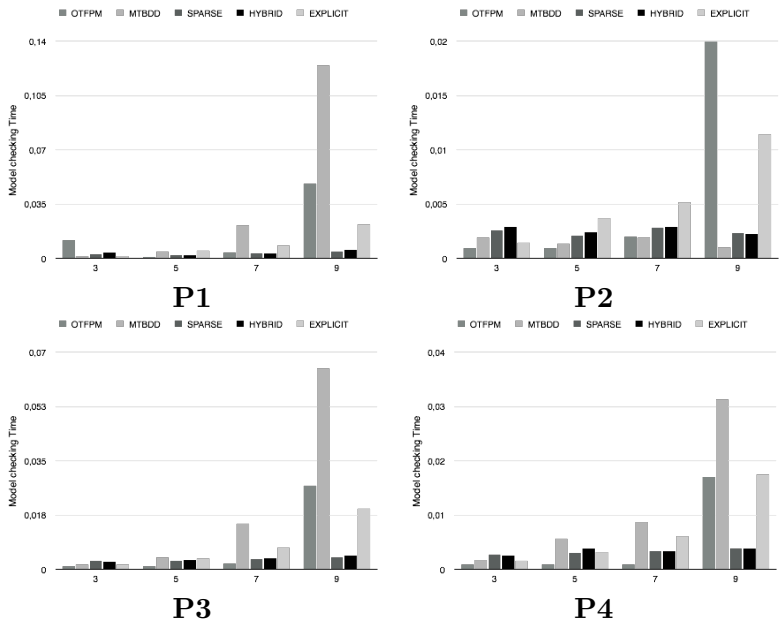

Figure 9: HSS: model checking execution times (in seconds) for ($N \in \{3, 5, 7, 9\}$)

	OTFPMC	MTBDD	SPARSE	HYBRID	EXPLICIT
P1					
$N = 11$	0.824	0.838	0.022	0.028	0.196
$N = 15$	32.744	52.345	1.545	1.607	∞
P2					
N=11	0.210	0.003	0.003	0.003	0.078
N=15	18.770	0.004	0.007	0.007	∞
P3					
N=11	0.369	0.368	0.016	0.020	0.146
N=15	14.006	23.083	1.046	1.069	∞
P4					
N=11	0.158	0.205	0.013	0.0137	0.112
N=15	18.205	13.778	0.805	0.621	∞

Table 11: HSS: model checking execution times (in seconds) for ($N \in \{11, 15\}$)

Processes	**States**	**Transitions**
3	99	392
5	128	2188
7	2283	13687
9	1122363	11223554
11	24811779	297741149
13	-	-
15	-	-

Table 12: RDP: size of considered models.

4.3.3 Randomised Dining Philosophers (RDP)

In the dining philosophers scenario n philosophers are sitting at a round table while performing only two activities: thinking and eating. In the middle of the table there is a large plate with spaghetti that is constantly refilled and between each pair of philosophers lies a chopstick. A philosopher needs both the chopstick on his right and his left at the same time to be able to eat. Pnueli and Zuck [25] describe a deadlock free distributed random algorithm. A philosopher picks the two chopsticks in random order. If he can get only one of the two chopsticks, he gives up eating (but may become hungry again later and try again). We consider a DTMC variant of the dining philosophers with n ranging from 3 to 15. In Table 12 we report the number of states and transitions of the considered instances of RDP.

For this case study we consider two properties: (**P1**) Philosopher 1 is the first to eat within the next 20 steps with probability greater than p while the other philosophers think; (**P2**) With probability greater than p, philosopher 1 is the first to eat while the other philosophers think[9]. We use the following formalisation in PCTL for a model with 10 philosophers:

(P1) $\mathcal{P}_{>p}((s2 = 0 \wedge s3 = 0 \wedge \cdots \wedge s10 = 0)\ \mathcal{U}^{\leq 20}\ s1 = 3)$

(P2) $\mathcal{P}_{>p}((s2 = 0 \wedge s3 = 0 \wedge \cdots \wedge s10 = 0)\ \mathcal{U}\ s1 = 3)$

where si denotes the state of the philosopher i, state $si = 0$ denotes that philosopher i is thinking and state $si = 3$ represents that philosopher i is eating. Both considered formulas capture a *conditional* reachability property, which is bounded in the case of **P1**.

Experimentation results In this scenario, the size of the state space (in terms of both number of states and transitions) increases exponentially with the number of philosophers. For this kind of models, when only a small subset of the state space has to be visited, our on-the-fly approach is very efficient. Indeed, for the 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 (see Figure 10). PRISM instead raises an *out-of-memory* exception for these cases for all PRISM engines. The execution times for the on-the-fly approach and the PRISM engines are shown in Fig. 10 for both properties **(P1)** and **(P2)**. For **(P1)** and a number of philosophers upto 5 the execution times for all methods are extremely small with respect to the time scale used in the figure. For 9 philosophers or more the explicit state engine of PRISM raises an *out-of-memory* exception and its results are not included in the table for those values. This is also the case for the results of **(P2)**.

4.3.4 SEIR computer epidemic model (SEIR).

We consider a model of a worm epidemic in a network of computers [24]. Each node in the network can be infected by a worm. Once a node is exposed, the worm remains latent for a while, and then activates

[9]Properties of single objects in the context of a larger population are relevant in e.g. population models [7].

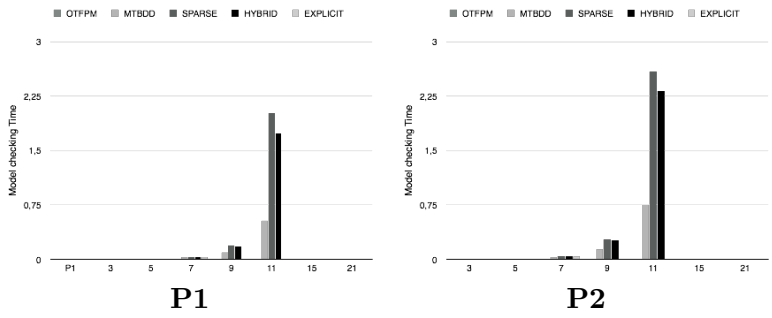

Figure 10: RDF: model checking execution times (in seconds). OTFPM takes a few milliseconds for all cases, whereas from 9 philosophers and above PRISM$_E$ runs out of memory, and the other PRISM engines run out of memory for 15 philosophers and above, so their results are not shown in the figure.

Processes	**States**	**Transitions**
3	64	512
5	1024	32768
7	16384	2097152
9	262144	134217728
11	1048576	1073741824

Table 13: SEIR: size of considered models.

so that the node can actively infect other nodes by sending them infected messages. After some time, an infected node can be patched, so that the infection is recovered. We assume that recovered nodes can become susceptible to infection again after a while, for example due to the appearance of a new version of the worm. Non-infected nodes may also be patched, but this event happens less frequently. Each node in the network can acquire infection from two sources, i.e. by the activity of a worm of an infected node or by an external source (for instance, by an email attachment received from outside the network). So, each node can be in one of four states: susceptible (S), exposed (E), actively infected (I), or recovered (R). We consider three different properties of an individual node in the context of the larger network: (**P1**) the probability that the first node is infected in the next 20 steps, while less then 25% of the nodes are exposed and none is infected, is at most p; (**P2**) the probability that the first node is eventually infected, while less then 25% of the nodes are exposed and none is infected, is at most p. We use the following formalisation in PCTL for a model with 4 nodes[10]:

(P1) $\mathcal{P}_{>p}((frac\_E < 0.25) \wedge (frac\_I = 0)\ \mathcal{U}^{\leq 20}\ s1 = 2)$

(P2) $\mathcal{P}_{>p}((frac\_E < 0.25) \wedge (frac\_I = 0)\ \mathcal{U}\ s1 = 2)$

Experimentation results Also in this scenario, like in the RDF considered in the previous section, only a small subset of the state space has to be visited to verify the properties. We have previously remarked that this situation fits well with our on-the-fly approach. However, thanks to the symmetry of the considered model, all the symbolic PRISM engines have comparable performance. This does not hold for PRISM$_E$ that, due to the large number of states and transitions, is not able to explicitly build the complete DTMC for 9 nodes or more. The results of the considered analyses are reported in Figure 11.

[10] *frac_E* and *frac_I* are PRISM functions denoting the fraction of nodes in state E and I respectively.

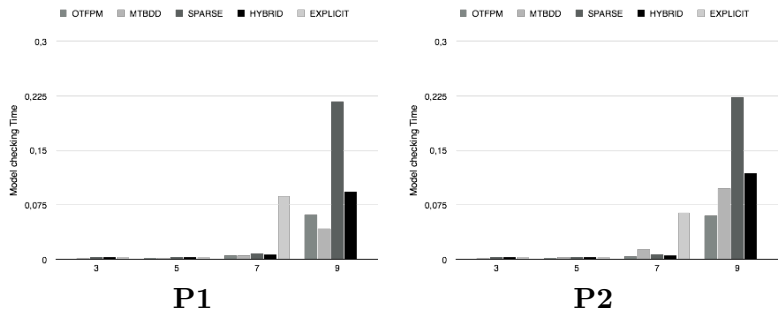

Figure 11: SEIR: model checking execution times (in seconds).

4.4 Concluding Remarks

Which of the two model checkers is more convenient to use depends on the problem at hand. For highly symmetric models PRISM is expected to perform significantly better because it can exploit powerful reduction techniques based on the underlying MTBDD structure. For models with much less symmetry the on-the-fly model checker has a performance that is comparable to that of the PRISM engines if the whole state space must be generated. As can be expected, the on-the-fly approach gives a large advantage in case of conditional reachability properties, in particular when the condition is such that a large part of the state space does not need to be generated.

The exploitation of the fact that the expansion phase of the on-the-fly model checker essentially generates a DTMC of a particular kind, namely a transient DTMC together with the efficient (both in memory and in computation time) computation of the probability to satisfy and *not* to satisfy the formula under analysis shows also in practical case studies to lead to accurate results in a number of iterations that can be estimated in advance. This is particularly relevant for relatively stiff models. This novel alternative solution method seems promising and could be considered as an alternative to the more standard Jacobi or Gauss-Seidel methods also in the case of global model-checkers such as PRISM.

Another successful technique that has been recently proposed to perform model checking of large scale systems is the *statistical model checker* [6]. Following this approach, a randomised algorithm is used to verify whether the considered specification satisfies a specific property with a certain degree of confidence. Indeed, the statistical model-checker is parameterised with respect to a given tolerance ε and error probability p. The used algorithm guarantees that the difference between the value computed by the algorithm and the exact one is greater than ε with a probability that is less than p. A detailed comparison between *statical* and *on-the-fly* model checking is out of the scope of this paper and will be investigated in the future. Here, we can remark that, in the general case, *statistical model checking* can be used only for the fragment of PCTL that does not contain *unbounded until*. Moreover, when the actual computed probability is close to the probability bound of a formula, methods based on statistical model checking may require that very many simulations need to be generated to be able to give a reliable answer. The statistical model checker included with PRISM can be used to perform some simple analysis in this direction. If we require a tolerance $\varepsilon = 10^{-6}$ and an error probability of 0.01, the satisfaction of formula **P1** for the RDP model of Section 4.3.3, with $p = 0.0002$, by the system composed of 15 philosophers needs $3,975,133$ simulation runs that are performed in 70 secs. Using the on-the-fly approach the verification takes only a few milliseconds.

Again, depending on the specific model and property of interest at hand, on-the-fly methods may be able to provide answers where other techniques do not. The experimental results reported in the current paper are meant to provide a more detailed insight in the strength and the limitations of the on-the-fly model checking approach in various settings. Of course, 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 with respect to the language and the specific semantic model of interest. The algorithm for unbounded until is new and is exploiting a 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 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 terms of state space and in terms of execution time. For the comparison the various engines of the probabilistic model checker PRISM have been used together with a number of benchmark case studies.

Overall, the on-the-fly method has been shown to have an execution time that is comparable with that of the PRISM engines for models that are not having specific characteristics such as strong forms of symmetry, that can be exploited by symbolic approaches, or for properties that do not give a particular advantage to on-the-fly techniques such as conditional reachability properties. However, in case of conditional reachability properties the on-the-fly approach may give results where other methods would generate a too large state space. It has been shown as an example that also in the case of statistical (or approximate) model checking there are cases in which an on-the-fly approach can be much faster than statistical methods. Of course, the combined availability of different model checking techniques provides most advantages.

The proposed algorithm for unbounded until properties has furthermore shown interesting advantages for the analysis of relatively stiff models in comparison to standard approaches in terms of a reliable and relatively efficient convergence to accurate probability values.

In related work by the authors the on-the-fly model checker has also been instantiated and used for fast mean field on-the-fly PCTL model checking for discrete time synchronous population models obtaining a scalability independent of the size of the population [7, 9]. Further work is planned on extensions of model checking techniques that concern spatial aspects of systems along the lines of recent work on spatial logic and model checking [28] as well as the application to a larger range of case-studies incorporating further probabilistic languages and related semantics. Further comparison and possible integration of statistical model checking techniques and on-the-fly techniques and the development of possible design and verification pathways is planned as well.

6 Acknowledgements

This research has been partially funded by the EU projects QUANTICOL (nr. 600708) and ASCENS (nr. 257414) and the IT MIUR project CINA.

References

[1] C. Courcoubetis, M. Y. Vardi, P. Wolper, M. Yannakakis, Memory-efficient algorithms for the verification of temporal properties, Form. Methods Syst. Des. 1 (2-3) (1992) 275–288. doi:10.1007/BF00121128.

[2] G. Bhat, R. Cleaveland, O. Grumberg, Efficient on-the-fly model checking for CTL*, in: LICS, IEEE Computer Society, 1995, pp. 388–397. doi:10.1109/LICS.1995.523273.

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

[4] H. Hansson, B. Jonsson, A logic for reasoning about time and reliability, Formal Aspects of Computing 6 (1994) 512–535. doi:10.1007/BF01211866.

[5] G. D. Penna, B. Intrigila, I. Melatti, E. Tronci, M. V. Zilli, Finite horizon analysis of markov chains with the murphi verifier, STTT 8 (4-5) (2006) 397–409. doi:10.1007/s10009-005-0216-7.

[6] H. L. S. Younes, M. Z. Kwiatkowska, G. Norman, D. Parker, Numerical vs. statistical probabilistic model checking: An empirical study, in: K. Jensen, A. Podelski (Eds.), Proc. 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04), Vol. 2988 of LNCS, Springer, 2004, pp. 46–60. doi:10.1007/978-3-540-24730-2_4.

[7] D. Latella, M. Loreti, M. Massink, On-the-fly fast mean-field model-checking, in: M. Abadi, A. Lluch-Lafuente (Eds.), Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers, Vol. 8358 of Lecture Notes in Computer Science, Springer, 2013, pp. 297–314. doi:10.1007/978-3-319-05119-2_17.

[8] J.-Y. Le Boudec, D. McDonald, J. Mundinger, A generic mean field convergence result for systems of interacting objects, in: QEST07, IEEE Computer Society Press, 2007, pp. 3–18, iSBN 978-0-7695-2883-0. doi:10.1109/QEST.2007.3.

[9] D. Latella, M. Loreti, M. Massink, On-the-fly PCTL fast mean-field model-checking for self-organising coordination - preliminary version, Tech. Rep. TR-QC-01-2013, Quanticol Technical Report, available on-line at http://www.quanticol.eu (2013).

[10] M. Kwiatkowska, G. Norman, D. Parker, PRISM 4.0: Verification of probabilistic real-time systems, in: G. Gopalakrishnan, S. Qadeer (Eds.), Proc. 23rd International Conference on Computer Aided Verification (CAV'11), Vol. 6806 of LNCS, Springer, 2011, pp. 585–591.

[11] D. Latella, M. Loreti, M. Massink, On-the-fly probabilistic model checking, in: I. Lanese, A. Lluch-Lafuente, A. Sokolova, H. T. Vieira (Eds.), Proceedings 7th Interaction and Concurrency Experience, ICE 2014, Berlin, Germany, 6th June 2014., Vol. 166 of EPTCS, 2014, pp. 45–59. doi:10.4204/EPTCS.166.6.
URL http://dx.doi.org/10.4204/EPTCS.166.6

[12] A. Pnueli, 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, 1977, pp. 46–57. doi:10.1109/SFCS.1977.32.

[13] K. Y. Rozier, M. Y. Vardi, LTL satisfiability checking, STTT 12 (2) (2010) 123–137. doi:10.1007/s10009-010-0140-3.

[14] G. J. Holzmann, The SPIN Model Checker - primer and reference manual, Addison-Wesley, 2004.

[15] S. Gnesi, F. Mazzanti, An abstract, on the fly framework for the verification of service-oriented systems, in: M. Wirsing, M. M. Hölzl (Eds.), Results of the SENSORIA Project, Vol. 6582 of LNCS, Springer, 2011, pp. 390–407. doi:10.1007/978-3-642-20401-2_18.

[16] C. Baier, B. R. Haverkort, H. Hermanns, J.-P. Katoen, Model-Checking Algorithms for Continuous-Time Markov Chains, IEEE Transactions on Software Engineering. IEEE CS 29 (6) (2003) 524–541. doi:10.1109/TSE.2003.1205180.

[17] M. Z. Kwiatkowska, G. Norman, D. Parker, Probabilistic symbolic model checking with PRISM: a hybrid approach, STTT 6 (2) (2004) 128–142. doi:10.1007/s10009-004-0140-2.

[18] A. Fernandez-Diaz, C. Baier, C. Benac-Earle, L.-A. Fredlund, Static partial order reduction for probabilistic concurrent systems, QEST 2012 0 (2012) 104–113. doi:10.1109/QEST.2012.22.

[19] C. Baier, P. R. D'Argenio, M. Grober, Partial order reduction for probabilistic branching time, Electr. Notes Theor. Comput. Sci. 153 (2) (2006) 97–116. doi:10.1016/j.entcs.2005.10.034.

[20] E. M. Hahn, H. Hermanns, B. Wachter, L. Zhang, Time-bounded model checking of infinite-state continuous-time Markov chains, Fundam. Inform. 95 (1) (2009) 129–155. doi:10.3233/FI-2009-145.

[21] D. Michie, Memo Functions and Machine Learning, Nature 218 (5136) (1968) 19–22.

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

[23] M. Kwiatkowska, G. Norman, D. Parker, The PRISM benchmark suite, in: Proc. 9th International Conference on Quantitative Evaluation of SysTems (QEST'12), IEEE CS Press, 2012, pp. 203–204.

[24] L. Bortolussi, J. Hillston, D. Latella, M. Massink, Continuous approximation of collective system behaviour: A tutorial, Performance Evaluation 70 (5) (2013) 317 – 349. doi:10.1016/j.peva.2013.01.001.
URL http://www.sciencedirect.com/science/article/pii/S0166531613000023

[25] A. Pnueli, L. Zuck, Verification of multiprocess probabilistic protocols, Distributed Computing 1 (1) (1986) 53–72.

[26] L. Helmink, M. Sellink, F. Vaandrager, Proof-checking a data link protocol, in: H. Barendregt, T. Nipkow (Eds.), Proc. International Workshop on Types for Proofs and Programs (TYPES'93), Vol. 806 of LNCS, Springer, 1994, pp. 127–165.

[27] T. Herman, Probabilistic self-stabilization, Inf. Process. Lett. 35 (2) (1990) 63–67. doi:10.1016/0020-0190(90)90107-9.

[28] V. Ciancia, D. Latella, M. Loreti, M. Massink, Specifying and verifying properties of space, in: J. Diaz, I. Lanese, D. Sangiorgi (Eds.), Theoretical Computer Science, Vol. 8705 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2014, pp. 222–235. doi:10.1007/978-3-662-44602-7_18.
URL http://dx.doi.org/10.1007/978-3-662-44602-7_18