Approximate Model Checking of Stochastic COWS

Paola Quaglia and Stefano Schivo

Dipartimento di Ingegneria e Scienza dell'Informazione, Università di Trento

Abstract. Given the description of a model and a probabilistic formula, approximate model checking is a verification technique based on statistical reasoning that allows answering whether or not the model satisfies the formula. Only a subset of the properties that can be analyzed by exact model checking can be attacked by approximate methods. These latest methods, though, being based on simulation and sampling have the advantage of not requiring the generation of the complete state-space of the model.
Here we describe an efficient tool for the approximate model checking of services written in a stochastic variant of COWS, a process calculus for the orchestration of services.

1 Introduction

Stochastic process calculi have been mainly defined to ground the formal quantitative analysis of both performance and reliability aspects of concurrent distributed systems. When specializing the distributed paradigm to the case of service-oriented computation, performance issues suddenly become even more demanding by clearly expressing features like, e.g., quality of service, resource usage, and dependability. That is why recent research work in the concurrency community focussed on the definition of probabilistic/stochastic extensions of calculi explicitly meant to specify services and hence natively equipped with primitives for rendering basic service operations as sessioning or protection [3, 15].

On the quantitative analysis side, the primary techniques to be applied are Monte Carlo simulation and probabilistic model checking. The first one, which is mostly used to reason about really huge systems (e.g., biological ones), consists in running a number of execution traces of the system, and then inferring the relevant information by applying statistical methods. Model checking, which can be either *exact* or *approximate*, grounds on a quite distinctive point of view. Both in the exact and in the approximate case, indeed, model checking consists in contrasting the behaviour of the system against a specific property expressed as a formula of a given probabilistic or stochastic logic (e.g., CSL [1]).

In more detail, exact model checking numerically checks the complete state-space of the system against a formula and returns a totally accurate result. Approximate model checking, instead, is based on simulation and sampling. The estimation if the given property holds is rather based on statistical reasoning on the generated samples. Different approaches offer a choice between *a priori* setting a fixed number of samples or not. In the first case, the generation of samples is stopped when the predetermined maximum is reached and an answer is given based on the available data. When the number

of samples is not fixed, the user can set a desired error probability (confidence level). The lower the confidence level, the bigger the number of samples to be generated.

Approximate model checking cannot have the same high level of accuracy as the result of the numerical methods of exact model checking. Also, given that samples are deemed to have finite length, approximate model checking cannot be used to check as many kinds of formulae as those checked by exact techniques. Building the complete state-space of the system, though, is not necessary. This is the good point of approximate model checking: memory requirements are negligible if compared to those imposed by numerical methods. Indeed, especially for loosely coupled interacting systems as those used for representing service computations, the size of the corresponding model typically suffers an exponential blow-up leading to state-space explosion.

This paper describes a tool for the approximate model checking of services described in Scows [17], the stochastic extension of COWS [14] (Calculus for Orchestration of Web Services), a calculus for service-oriented computation strongly inspired by WS-BPEL. We first overview the source language (Section 2) which extends the work presented in [15] to polyadic communication. Then we describe the tool, called Scows_amc, together with the foundational theory it is based upon (Section 3). Further, in Section 4, based on a simple service describing the classical scenario of the dining philosophers, the efficiency of Scows_amc is compared to the results obtained by applying Scows_lts [2], a tool that builds the complete Continuos Time Markov Chain (CTMC) corresponding to Scows services and allows their exact model checking through PRISM [13].

2 Scows overview

Scows is a stochastic extension of COWS, to which the capability to represent quantitative aspects of services (in particular, execution time) has been added. The semantics of Scows reflects the one of the basic calculus, where an original communication paradigm is used. This paradigm is based on a mechanism of *best-matching* of the parameters of complementary invoke (send) and request (receive) activities.

The COWS communication paradigm is best illustrated by a simple example. To this purpose, let us consider the following term composed of three parallel services:

$$[\, n_1, n_2, y_1, y_2 \,](p.o \,!\, \langle n_1, n_2 \rangle \mid p.o \,?\, \langle n_1, y_2 \rangle .\, s_1 \mid p.o \,?\, \langle y_1, y_2 \rangle .\, s_2) \tag{1}$$

where $[\, n_1, n_2, y_1, y_2 \,]$ is a scope delimiter for n_1, n_2, y_1, and y_2. The leftmost subcomponent $p.o \,!\, \langle n_1, n_2 \rangle$ can send the tuple of names $\langle n_1, n_2 \rangle$ over the endpoint $p.o$ (an endpoint is given by a *partner name* and an *operation name*, respectively p and o in this case). In (1) the middle service $p.o \,?\, \langle n_1, y_2 \rangle .\, s_1$ is a request-guarded term. It is waiting to receive over the endpoint $p.o$ a pair of actual parameters matching the formal tuple $\langle n_1, y_2 \rangle$, where y_2 is a variable. Names and variables play quite distinctive roles in the matching policy. Names can be thought of as constants or ground objects: each name can only match itself. Variables instead can match any name. Briefly put, matching two tuples means finding a substitution of names for variables that, applied to the tuple of the receiving service, makes it equal to the tuple of actual parameters offered by the

sending service. So, for instance, the tuples $\langle n_1, n_2\rangle$ and $\langle n_1, y_2\rangle$ match because the substitution of n_2 for y_2, written $\{^{n_2}/_{y_2}\}$, when applied to $\langle n_1, y_2\rangle$ results in $\langle n_1, n_2\rangle$. Going back to the behaviour of the service $p.o\,?\langle n_1, y_2\rangle.\, s_1$, the execution of the request activity would unblock the continuation service s_1 and would make it dependent on the substitution induced by the matching. For instance, a communication between the leftmost parallel services in (1) would result in running $s_1\{^{n_2}/_{y_2}\}$. The rightmost parallel component $p.o\,?\langle y_1, y_2\rangle.\, s_2$ is much similar in its structure to the middle one. It is a request-guarded term with potentials for interaction with whichever service can offer over $p.o$ a tuple matching $\langle y_1, y_2\rangle$, where y_1 is yet another variable. The tuple $\langle n_1, n_2\rangle$ offered by the leftmost service matches $\langle y_1, y_2\rangle$ by inducing the substitution $\{^{n_1}/_{y_1}, ^{n_2}/_{y_2}\}$. Out of a set of potential communications, the *best-matching* mechanism adopted by COWS amounts to allow only those communications that induce substitutions as small as possible. So in our example $p.o\,!\langle n_1, n_2\rangle$ can only communicate with $p.o\,?\langle n_1, y_2\rangle.\, s_1$. Here notice that if the global service were augmented by adding a fourth parallel component $p.o\,?\langle y_1, n_2\rangle.\, s_3$, then $p.o\,!\langle n_1, n_2\rangle$ could communicate with either $p.o\,?\langle n_1, y_2\rangle.\, s_1$ or $p.o\,?\langle y_1, n_2\rangle.\, s_3$. So the best-matching policy is not a cure against non-determinism. It rather serves the purpose of implementing sessioning: opening a session is rendered by passing a unique name as session identifier, and all the communications relative to that session will carry on that identifier as parameter. Moreover, in the same way as the interaction of $p.o\,!\langle n_1, n_2\rangle$ with $p.o\,?\langle n_1, y_2\rangle.\, s_1$ pre-empties the communication with $p.o\,?\langle y_1, y_2\rangle.\, s_2$, concluding the operations relative to an open session will have priority over opening a new session (just think of n_1 as of a session identifier).

In the stochastic extension of COWS, each basic action (the two communicating primitives request and invoke, plus a special killing activity used for process termination) is enriched by a real number representing the rate of an exponential distribution, which models the time taken by the service to execute the corresponding action. For example, letting $\delta_1, \delta_2, \delta_3$ to stay for rates, the term in (1) would be written as follows in Scows:

$$[\,n_1, n_2, y_1, y_2\,]\,(\,(p.o!\langle n_1, n_2\rangle, \delta_1)\mid (p.o?\langle n_1, y_2\rangle, \delta_2).\, s_1 \mid (p.o?\langle y_1, y_2\rangle, \delta_3).\, s_2\,).$$

The interaction paradigm adopted in the basic calculus, and retained in Scows, is responsible of a rather complex rate computation method for communications. Indeed, the competition induced by the best-matching policy for pairing invoke and request activities has to be correctly reflected in the rate of the resulting action. In particular, polyadicity makes rate computation quite more intricate in Scows than what can be seen in [15] where a monadic version of the basic calculus was considered. On the other hand, polyadic communication allows the user to explicitly render the use of session identifiers.

We will briefly illustrate the main issues about rate computation by means of a simple example. Consider the following service definition:

$$S = [\,m, n, n', x, y\,]\,(\underbrace{(p.o!\langle m, n\rangle, \delta_1)}_{s_1} \mid \underbrace{(p.o!\langle m, n'\rangle, \delta_2)}_{s_2} \mid \underbrace{(p.o!\langle n, n'\rangle, \delta_3)}_{s_3} \mid \underbrace{(p.o!\langle n, n\rangle, \delta_4)}_{s_4}$$
$$\mid \underbrace{(p.o?\langle m, x\rangle, \gamma_1).\,\mathbf{0}}_{s_5} \mid \underbrace{(p.o?\langle y, n'\rangle, \gamma_2).\,\mathbf{0}}_{s_6})$$

Using the notation $S_j \rhd S_k$ to mean that services S_j and S_k communicate (with S_j sending and S_k receiving), we list below all the communications allowed in S by the matching paradigm:

- $S_1 \rhd S_5$, matching m with m and substituting x by n;
- $S_2 \rhd S_5$, matching m with m and substituting x by n';
- $S_2 \rhd S_6$, substituting y by m and matching n' with n';
- $S_3 \rhd S_6$, substituting y by n and matching n' with n';
- no service can successfully match S_4's tuple, so S_4 does not communicate.

Observing the interaction capabilities of the processes in the example, we suggest an asymmetric way to intend the communication paradigm in Scows in the sense that the choice of an invoke action also determines the set of possible communications. For instance, one single communication is possible when choosing the invoking service S_1, and the same holds when selecting S_3. On the other hand, if the sending service S_2 is chosen, then there are two possible communications: $S_2 \rhd S_5$ and $S_2 \rhd S_6$. Note that, as both $S_2 \rhd S_5$ and $S_2 \rhd S_6$ induce one substitution, they are equally viable from the non-stochastic point of view, while the stochastic rates of the two transitions can determine different probabilities for the execution of the two distinct actions.

In case of communications, the rate computation also depends on the so-called *apparent rate* of the participants. The apparent rate represents the rate at which actions of the same type are perceived by an observer located outside the service. We consider two actions to be indistinguishable from the point of view of an external observer if these actions are competing to participate with the same role in the same communication. In the case of an invoke action the actions competing with it are all the send-actions available on the same endpoint. The case of a request action is more complicated, as it requires to take into account also all the actions which would have been in competition with the chosen one if another invoke action would have been selected. The formal definition of the Scows operational semantics is outside the scope of the present paper. For more details on the actual computation of apparent rates the interested reader is referred to [17].

The rate of a communication event is obtained by multiplying the apparent rate of the communication by the probability to choose exactly the two participants involved in that communication. Adopting a classical way of approximating exponential rates [9], we take the apparent rate of a communication to be the minimum between the apparent rates of the participating services (i.e., the communication proceeds at the speed of the "slowest" of the participants). So, the formula for the rate of a communication between the invoking service S_j and the requesting service S_k has the following form:

$$\mathcal{P}(S_j \rhd S_k) \cdot \mathtt{min}\Big(\mathtt{appRate}(S_j), \mathtt{appRate}(S_k)\Big),$$

where $\mathcal{P}(S_j \rhd S_k)$ is the probability that services S_j and S_k are involved in the communication, and $\mathtt{appRate}(S_j)$ (resp. $\mathtt{appRate}(S_k)$) represents the apparent rate of the invoke (request) action computed considering the whole service containing S_j and S_k. As we consider request actions to be dependent on invoke actions, the probability to choose a particular invoke-request pair is computed as a conditional probability:

$$\mathcal{P}(S_j \rhd S_k) = \mathcal{P}(S_j) \cdot \mathcal{P}(S_k \mid S_j).$$

This means that the probability of a communication between the invoke S_j and the request S_k is given by the product of the probability to have chosen S_j among all possible invoke actions on endpoint $p.o$ and the probability to choose S_k among the request actions made available by the choice of S_j (i.e., the request actions best-matching S_j).

In the above example, the probability that a communication occurs between S_2 and S_5 is calculated as follows:

$$\begin{aligned}\mathcal{P}(S_2 \rhd S_5) &= \mathcal{P}(S_2) \cdot \mathcal{P}(S_5 \mid S_2) \\ &= \frac{\delta_2}{\delta_1 + \delta_2 + \delta_3} \cdot \frac{\gamma_1}{\gamma_1 + \gamma_2}\,.\end{aligned}$$

Given that S_4 cannot take part into any communication, the rate δ_4 of service S_4 is not taken into account. On the other hand, as S_5 is the single request action matching with S_1, the probability of a communication between S_1 and S_5 is basically the probability of choosing S_1:

$$\begin{aligned}\mathcal{P}(S_1 \rhd S_5) &= \mathcal{P}(S_1) \cdot \mathcal{P}(S_5 \mid S_1) \\ &= \frac{\delta_1}{\delta_1 + \delta_2} \cdot 1\,.\end{aligned}$$

Here notice that we do not take into account the rate δ_3 of service S_3, as S_3 cannot communicate with S_5 ($n \neq m$) and thus cannot influence the above communication.

3 Scows_amc

Generating a CTMC from a Scows term, as required by exact model checking, can be a computationally costly task, and could even lead to state-explosion when building the underlying transition system. This issue is most evident when a model comprises a number of loosely coupled components, as it is often the case when dealing with distributed systems. A compositional generation of the transition system might help minimizing the state space, thanks to the fact that parallel components could be considered in isolation and then merged with less-than-exponential execution time. Unfortunately, this type of approach cannot be applied in the case of Scows. This is due to the adopted communication paradigm which requires the complete knowledge of the model to calculate each single communication action, de facto preventing a compositional generation of the transition system. In languages with multi-way synchronization and not featuring name-passing, like e.g. in PEPA [9], the compositional approach can be applied and is in fact feasible: for instance, such an approach is used for the generation of CTMCs from PEPA models in PRISM, which is based on MTBDD (Multi-Terminal Binary Decision Diagrams [4, 8]) representations. Another example of application of the same principle can be seen in CASPA [12], a tool which generates a MTBDD representation from YAMPA, a stochastic process algebra based on TIPP [7].

Below we present a tool, called Scows_amc, that allows statistical model checking of Scows terms while maintaining acceptable computation time and approximation values. In order not to generate complete transition systems, we base our approach on direct simulations of Scows models. In particular, we generate a number of simulation

traces by applying the operational semantics rules directly to Scows services, and then perform the computations necessary to check the desired formula against these traces. As a single execution trace of a stochastic model is by definition a random walk on the transition system of the model, we resort to statistical reasoning in order to estimate the size of the error we make in evaluating the requested property through a finite number of random walks. The theories behind the reasoning on which the approach used by Scows_amc is based are the one adopted in Ymer [11], and the one adopted in APMC [5, 6]. In particular, letting $\bowtie \in \{<, \leqslant, >, \geqslant\}$, $t_0, t_1 \in \mathbb{R}^+$, and $\theta \in [0, 1]$, Scows_amc can model check Scows terms against the usual CSL time-bounded until properties of the form:

$$\mathcal{P}_{\bowtie\theta}\,[\Psi_1\ \mathcal{U}^{[t_0,t_1]}\ \Psi_2] \tag{2}$$

and their numerical corresponding in the following shape:

$$\mathcal{P}_{=?}\,[\Psi_1\ \mathcal{U}^{[t_0,t_1]}\ \Psi_2] \tag{3}$$

which can be read as:

> "Is there a probability $p \bowtie \theta$ that state formula Ψ_1 will hold until, inside the time interval $[t_0, t_1]$, state formula Ψ_2 holds?"

and, respectively,

> "What is the probability that state formula Ψ_1 will hold until, inside the time interval $[t_0, t_1]$, state formula Ψ_2 holds?"

where the state formulae Ψ_1 and Ψ_2 are to be intended as state labels.

The truth value of CSL probabilistic formulae of the type (2) is calculated through the *sequential probability ratio test* [18]. This method requires to perform a sequence of observations of the hypothesis to be tested. After each observation an error estimation is made, taking into account the results of all the previous observations. When a given error threshold is crossed, the hypothesis is either accepted or rejected. In our case, performing an observation corresponds to testing the formula over an execution trace, which is generated on demand. This kind of approach does not require to have an exact estimation of the real probability to have the property verified: it only checks whether the probability lies below or beyond the specified threshold. The algorithm implemented to apply the sequential probability ratio test and to evaluate formulae of type (2) is reported as pseudocode in Algorithm 1. The applied method computes estimations based on three approximation parameters: α (the probability to get a false negative), β (the probability to get a false positive), and δ (the semi-distance from θ used to define the indifference region). These parameters must be chosen making a trade-off between the execution time and the answer confidence.

In order to obtain the approximate model checking of formulae of type (3), we rely on a method presented in [5] and based on [10]. As in the case of formulae of type (2), the value of properties is estimated by means of a series of observations on random walks over the transition system. This time, however, the number of observations necessary to obtain the desired approximation level is determined *before* observations are made. This allows the user to make trade-offs between speed and approximation with

Algorithm 1 The algorithm used to perform the sequential probability ratio test.

input $\alpha, \beta, \delta, \Phi = \mathcal{P}_{\bowtie\theta}[\varphi]$
$p_0 \leftarrow \theta + \delta$
$p_1 \leftarrow \theta - \delta$
$logA \leftarrow \ln \frac{1-\beta}{\alpha}$
$logB \leftarrow \ln \frac{\beta}{1-\alpha}$
$nSamples \leftarrow 0$
$d \leftarrow 0$
while $logB < d \wedge d < logA$ **do**
  generate a random walk σ
  if $\sigma \models \varphi$ **then**
    $d \leftarrow d + \ln \frac{p_1}{p_0}$
  else
    $d \leftarrow d + \ln \frac{1-p_1}{1-p_0}$
  end if
  $nSamples \leftarrow nSamples + 1$
end while
if $\bowtie \in \{>, \geqslant\}$ **then**
  return $d \leqslant logB$
else
  return $d > logB$
end if

a deeper insight on the effects of his choices. Two main parameters are used for error estimation: the *approximation parameter* ε, and the *confidence parameter* δ. It can be shown that the evaluation of the given formula on $O\left(\frac{1}{\varepsilon^2} \cdot \log \frac{1}{\delta}\right)$ random walks brings to a probability estimation differing from the real value by less than ε with probability $1 - \delta$. The number of random walks necessary to obtain the desired result is given by the following formula:

$$nObservations = 4 \cdot \frac{\log \frac{2}{\delta}}{\varepsilon^2}. \tag{4}$$

Algorithm 2 is used in Scows_amc for the estimation of type (3) formulae and it is in fact the one presented in [5]. The idea on which the algorithm is based in order to compute the probability estimation for $\mathcal{P}_{=?}[\varphi]$ is to execute a fixed number of observations of the truth value of the formula φ, and then count the number of positive results. The probability that φ is true is given by the ratio between the number of positive observations and the total number of observations.

A final observation is about what is involved in checking until path formulae. Algorithm 3 is used to obtain the truth value for one of such formulae on a single simulation trace of the model. Notice that the algorithm performs the checking of the formula as the generation of the trace goes along. This is a solution which has a better average execution time w.r.t. an approach in which a complete simulation trace is generated and then checked against the path formula. This is possible thanks to the fact that the until formula has time bounds, which allows us to stop the generation of a trace when "time is up" or when a truth value for the formula has been found, even if the simulation could proceed further.

Algorithm 2 The algorithm used in Scows_amc for the estimation of a CSL formula in the form $\mathcal{P}_{=?}[\varphi]$.

input δ, ε
$nObservations \leftarrow 4 \log\left(\frac{2}{\delta}\right)/\varepsilon^2$
$count \leftarrow 0$
for $i = 1$ to $nObservations$ **do**
  generate a random walk σ
  if $\sigma \models \varphi$ **then**
    $count \leftarrow count + 1$
  end if
end for
return $count/nObservations$

Algorithm 3 Verification of a bounded until formula.

input α, β, δ, Scows*model*, $\varphi = \Phi_1\ \mathcal{U}^{[tmin,tmax]}\ \Phi_2$
$totalTime \leftarrow 0$
$nextTime \leftarrow 0$
$currState \leftarrow$ initialState(Scows*model*)
$transitions \leftarrow$ computeTransitions($currState$)
while ¬isEmpty($transitions$) **do**
  ($nextState$, τ) $\leftarrow$ computeNextState($transitions$)
  $nextTime \leftarrow totalTime + \tau$
  if $tmin \leqslant totalTime$ **then**
    if verify(Φ_2, $currState$, α, β, δ) **then**
      return `true`
    else if ¬verify(Φ_1, $currState$, α, β, δ) **then**
      return `false`
    end if
  else
    if ¬verify(Φ_1, $currState$, α, β, δ) **then**
      return `false`
    else if $tmin < nextTime \wedge$ verify(Φ_2, $currState$, α, β, δ) **then**
      return `true`
    end if
  end if
  $currState \leftarrow nextState$
  $totalTime \leftarrow nextTime$
  if $tmax < totalTime$ **then**
    return `false`
  end if
  $transitions \leftarrow$ computeTransitions($currState$)
  if isEmpty($transitions$) **then**
    return verify(Φ_2, $currState$, α, β, δ)
  end if
end while
return `false`

4 Comparison with CTMC generation

To the best of our knowledge, the single other tool available for the quantitative model checking of Scows is Scows_lts [2], which takes an approach orthogonal to that of Scows_amc. Indeed Scows_lts performs probabilistic model checking by generating a CTMC from a Scows model and then by using PRISM to obtain an assessment of the model.

In what follows, we show a test for the performance of Scows_amc by comparing it against Scows_lts. As a test-bed for the comparison we use a simple model of the system of the dining philosophers, an instance of which is presented in Table 1 using the Scows syntax accepted by Scows_amc. We model the problem using knives and forks as cutlery, and coupling right-handed philosophers with left-handed ones, in order to have an alternating pattern allowing each philosopher to eat with the preferred hand. The rates `r1`, `r2`, `r3`, ... are defined as parameters of the model. This feature is meant to foster sensitivity analysisis, and indeed Scows_lts can itself handle parametric rates.

The model in Table 1 is used in the experiment varying the number of dining philosophers between 2 and 12, properly adapting the available cutlery. As we differentiate between right-handed and left-handed philosophers, the resulting models will include an even number of philosophers. The CSL formula against which all the models are checked is the following one:

$$\mathcal{P}_{=?}\,[\texttt{true}\ \mathcal{U}^{[0,\texttt{T}]}\ \texttt{fed} = \texttt{N}]$$

meaning

> "What is the probability that `N` philosophers are fed at time `T`?".

The formula has been checked with parameters `N` and `T` varying between 0 and the number of philosophers in the model, and, respectively, between 0 and 40 time units. The settings for Scows_amc have been chosen so to obtain results with approximation of 10^{-2} with confidence of 0.9, and hence are $\varepsilon = 0.01$ and $\delta = 0.1$. This implies that 52042 simulation traces need to be computed for each configuration of the CSL formula. Actually, the number of computed traces is 52042 in total, as our tool uses an optimization which allows us to reduce the number of simulation traces needed to obtain the requested evaluations. Basically, we reuse the same trace for testing all the formulae which we need to test before proceeding with the computation of a new simulation trace.

As said, Scows_amc is compared against Scows_lts, which actually exploits PRISM for the analysis of CTMCs. The version of PRISM employed in our tests is the 3.3.1, available for download from the PRISM web site [16]. The CSL property has been checked against the relevant CTMCs by means of both the numerical (exact) and the simulation-based (approximate) approaches available in PRISM. As the approximate model checking used in PRISM is the same as the one used in Scows_amc, we have used the same parameters also for PRISM. All the tests have been executed on a workstation equipped with an Intel (R) Pentium (R) D 3.40 GHz CPU and 2 Gigabyte of RAM, running Ubuntu Linux 9.10.

The results of the comparison are shown in Table 2, where we report the execution time of Scows_amc against that of Scows_lts together with PRISM, when checking

```
//Agents

//RHphil: right-handed philosopher
RHphil(right#, left#) =
[fork][knife]( (right#.take#?<fork>,r1) . (left#.take#?<knife>,r2) .
    [eat#][food#]( (eat#.eat#!<food#>,r3) | (eat#.eat#?<food#>,r4) .
         ( (left#.release#!<knife>,r5) | (right#.release#!<fork>,r6) )
    )
);

//LHphil: left-handed philosopher
LHphil(right#, left#) =
[knife][fork]( (right#.take#?<knife>,r7) . (left#.take#?<fork>,r8) .
    [eat#][food#]( (eat#.eat#!<food#>,r9) | (eat#.eat#?<food#>,r10) .
         ( (left#.release#!<fork>,r11) | (right#.release#!<knife>,r12) )
    )
);

Cutlery(f#) = [p#] ( (f#.take#!<p#>,r13)
                   | (f#.release#?<p#>,r14).Cutlery(f#) );

$
//initial process

[fork1#][knife1#][fork2#][knife2#][take#][release#] (
      RHphil(fork1#, knife1#) | LHphil(knife1#, fork2#)
    | RHphil(fork2#, knife2#) | LHphil(knife2#, fork1#)
    | Cutlery(fork1#) | Cutlery(knife1#)
    | Cutlery(fork2#) | Cutlery(knife2#)
)

$

//counter definitions

fed : [ 0 .. 4 ];

$
//cows actions <-> counter modifications

eat#.eat#<*>: fed < 4 : (fed' = fed + 1);
```

Table 1. An instance of the dining philosophers system modelled in Scows.

Philosophers	State space size	Scows_lts	PRISM		Scows_amc
			Exact	Approx.	
2	20	0.9	1.9	95.5	395.6
4	249	345.4	153.5	1871.2	5537.8
6	3247	523173.0	138749.0	73729.4	31109.4
8	-	-	-	-	113603.0
10	-	-	-	-	309769.1
12	-	-	-	-	719487.9

Table 2. Computational time results for the example model checking (time is expressed in seconds). The two columns for PRISM correspond to the numerical (exact) and simulation-based (approximate) model checking approaches.

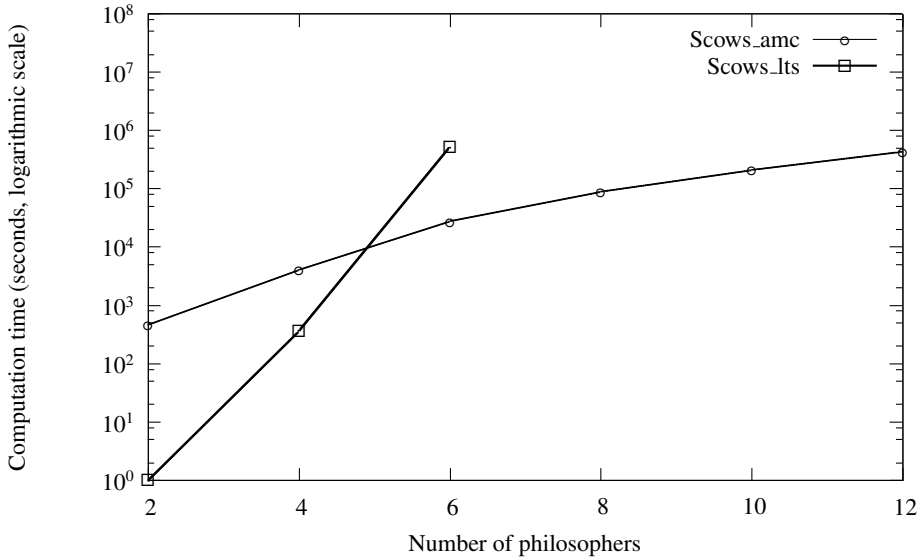

Fig. 1. Graph plotting execution time performances of the two approaches for model checking Scows models.

the models against the CSL formula. The time taken to model check the CTMC with PRISM is shown separately in order to highlight the actual time taken to produce the relevant CTMC. The execution time results are plotted in Figure 1. Data for 8, 10, and 12 philosophers are not available for the case of CTMC-based model checking, as the estimated computational times for the generation of the CTMCs was too high. Relative to this issue, we notice here that Scows_lts undergoes a number of computationally heavy optimizations mainly related to congruence checking of the states of the labelled transition system the CTMC is based upon. These optimizations, which involve, e.g., verifying α-equivalence of Scows terms, are fundamental to keep the state-space as small as possible and hence to limit the effects of memory usage.

Our comparison in Table 2 shows that when the number of states in the model is anticipated to be at most in the order of hundreds, the most convenient approach to model checking is to build a CTMC from the Scows model and use CTMC-based tools to perform the desired performance measures (either exact or approximate). Conversely, when the size of the state-space is estimated to be larger than few hundreds, the execution time of Scows_amc is expected to be lower. This execution speed comes at the price of precision, which however can be adjusted as necessity dictates.

5 Concluding remarks

A tool for the approximate model checking of Scows was described. An application of the tool was also presented, measuring its performances in terms of execution time. The example clearly shows, whenever applicable, the advantage of approximated model checking over its exact counter-part which involves the generation of the full state-space of the term.

Acknowledgements This work has been partially sponsored by the project SENSORIA, IST-2005-016004.

References

1. Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert Brayton. Model-checking continous-time Markov chains. *ACM Trans. on Computational Logic*, 1(1):162–170, 2000.
2. Igor Cappello and Paola Quaglia. A Tool for Checking Probabilistic Properties of COWS Services. Proceedings of TGC 2010. Tool available at `http://disi.unitn.it/~cappello/`, 2010.
3. Rocco De Nicola, Diego Latella, Michele Loreti, and Mieke Massink. MarCaSPiS: a Markovian Extension of a Calculus for Services. *Electronic Notes in Theoretical Computer Science*, 229(4):11–26, 2009.
4. Masahiro Fujita, Patrick C. McGeer, and Jerry C.-Y. Yang. Multi-Terminal Binary Decision Diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10:149–169(21), 1997.
5. Thomas Hérault, Richard Lassaigne, Frédéric Magniette, and Sylvain Peyronnet. Approximate probabilistic model checking. In *Proc. 5th Int. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI 2004)*, volume 2937 of *LNCS*, pages 307–329. Springer, 2004.

6. Thomas Hérault, Richard Lassaigne, and Sylvain Peyronnet. APMC 3.0: Approximate Verification of Discrete and Continuous Time Markov Chains. In *Proc. 3rd Int. Conf. on Quantitative Evaluation of Systems, QEST 2006*, pages 129–130. IEEE, 2006.
7. Holger Hermanns, Ulrich Herzog, and Joost-Pieter Katoen. Process algebra for performance evaluation. *Theoretical Computer Science*, 274(1-2):43 – 87, 2002.
8. Holger Hermanns, Joachim Meyer-Kayser, and Markus Siegle. Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In B. Plateau, W. Stewart, and M. Silva, editors, *Proc. 3rd International Workshop on Numerical Solution of Markov Chains (NSMC 1999)*, pages 188–207. Prensas Universitarias de Zaragoza, 1999.
9. Jane Hillston. *A compositional approach to performance modelling*. Cambridge University Press, New York, NY, USA, 1996.
10. Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.
11. Håkan L. Younes. *Verification and Planning for Stochastic Processes with Asynchronous Events*. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2005.
12. Georg Wolfgang Matthias Kuntz. *Symbolic Semantics and Verification of Stochastic Process Algebras*. PhD thesis, Friedrich-Alexander-Universitaet Erlangen-Nuernberg, Erlangen, March 2006.
13. Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic Model Checking for Performance and Reliability Analysis. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):40–45, 2009.
14. Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. Calculus for Orchestration of Web Services. In *Proc. ESOP 2007*, volume 4421 of *LNCS*, pages 33–47, 2007. Full version available at `http://rap.dsi.unifi.it/cows/`.
15. Davide Prandi and Paola Quaglia. Stochastic COWS. In B.J. Krämer, K.-J. Lin, and P. Narasimhan, editors, *Proc. 5th International Conference on Service Oriented Computing, ICSOC 2007*, volume 4749 of *LNCS*, pages 245–256. Springer, 2007.
16. PRISM homepage. `http://www.prismmodelchecker.org/`.
17. Stefano Schivo. *Statistical model checking of Web Services.* PhD thesis, Int. Doctorate School in Information and Communication Technologies, University of Trento, 2010.
18. Abraham Wald. Sequential tests of statistical hypotheses. *The Annals of Mathematical Statistics*, 16(2):117–186, 1945.