On-the-Fly Model Checking for Extended Action-Based Probabilistic Operators

Radu Mateescu and José Ignacio Requeno

Inria
CNRS, LIG, F-38000 Grenoble, France
Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France

Abstract. The quantitative analysis of concurrent systems requires expressive and user-friendly property languages combining temporal, data-handling, and quantitative aspects. In this paper, we aim at facilitating the quantitative analysis of systems modeled as PTSs (*Probabilistic Transition Systems*) labeled by actions containing data values and probabilities. We propose a new regular probabilistic operator that computes the probability measure of a path specified by a generalized regular formula involving arbitrary computations on data values. This operator, which subsumes the Until operators of PCTL and their action-based counterparts, can provide useful quantitative information about paths having certain (e.g., peak) cost values. We integrated the regular probabilistic operator into MCL (*Model Checking Language*) and we devised an associated on-the-fly model checking method, based on a combined local resolution of linear and Boolean equation systems. We implemented the method in the EVALUATOR model checker of the CADP toolbox and experimented it on realistic PTSs modeling concurrent systems.

1 Introduction

Concurrent systems, which are becoming ubiquitous nowadays, are complex software artifacts involving qualitative aspects (e.g., concurrent behaviour, synchronization, data communication) as well as quantitative aspects (e.g., costs, probabilities, timing information). The rigorous design of such systems based on formal methods and model checking techniques requires versatile temporal logics able to specify properties about qualitative and quantitative aspects in a uniform, user-friendly way. During the past two decades, a wealth of temporal logics dealing with one or several of these aspects were defined and equipped with analysis tools [8, 3]. One of the first logics capturing behavioral, discrete-time, and probabilistic information is PCTL (*Probabilistic CTL*) [17].

In this paper, we propose a framework for specifying and checking temporal logic properties combining actions, data, probabilities, and discrete-time on PTSs (*Probabilistic Transition Systems*) [22], which are suitable models for representing value-passing concurrent systems with interleaving semantics. In PTSs, transitions between states are labeled by actions that carry, in addition to probabilistic information, also data values sent between concurrent processes during handshake communication. Our contributions are twofold.

Regarding the specification of properties, we propose a new regular probabilistic operator, which computes the probability measure of a path (specified as a regular formula on actions) in a PTS. Several probabilistic logics have been proposed in the action-based setting. PML (*Probabilistic Modal Logic*) [22] is a variant of HML with modalities indexed by probabilities, and was introduced as a modal characterization of probabilistic bisimulation. GPL (*Generalized Probabilistic Logic*) [9] is a probabilistic variant of the alternation-free modal μ-calculus, able to reason about execution trees, and equipped with a model checking algorithm relying on the resolution of non-linear equation systems. Compared to these logics, our probabilistic operator is a natural (action-based) extension of the Until operator of PCTL: besides paths of the form $a^*.b$ (the action-based counterpart of Until operators), we consider more general paths, specified by regular formulas similar to those of PDL (*Propositional Dynamic Logic*) [14]. To handle the data values present on PTS actions, we rely on the regular formulas with counters of MCL (*Model Checking Language*) [28], which is an extension of first-order μ-calculus with programming language constructs. Moreover, we enhance the MCL regular formulas with a generalized iteration operator parameterized by data values, thus making possible the specification of arbitrarily complex paths in a PTS.

Regarding the evaluation of regular probabilistic formulas on PTSs, we devise an on-the-fly model checking method based on translating the problem into the simultaneous local resolution of a linear equation system (LES) and a Boolean equation system (BES). For probabilistic operators containing dataless MCL regular formulas, the sizes of the LES and BES are linear (resp. exponential) w.r.t. the size of the regular formula, depending whether it is deterministic or not. In the action-based setting, the determinism of formulas is essential for a sound translation of the verification problem to a LES. For general data handling MCL regular formulas, the termination of the model checking procedure is guaranteed for a large class of formulas (e.g., counting, bounded iteration, aggregation of values, computation of costs over paths, etc.) and the sizes of the equation systems depend on the data parameters occurring in formulas. It is worth noticing that on-the-fly verification algorithms for PCTL were proposed only recently [23], all previous implementations, e.g., in PRISM [20] having focused on global algorithms. Our method provides on-the-fly verification for PCTL and its action-based variant PACTL, and also for PPDL (*Probabilistic PDL*) [19], which are subsumed by the regular probabilistic operator of MCL. We implemented the method in the EVALUATOR [28] on-the-fly model checker of the CADP toolbox [16] and experimented it on various examples of value-passing concurrent systems.

The paper is organized as follows. Section 2 defines the dataless regular probabilistic operator and Section 3 presents the on-the-fly model checking method. Section 4 is devoted to the data handling extensions. Section 5 briefly describes the implementation of the method within CADP and illustrates it for the quantitative analysis of mutual exclusion protocols. Finally, Section 6 gives concluding remarks and directions of future work.

2 Dataless Regular Probabilistic Operator

As interpretation models, we consider PTSs (*Probabilistic Transition Systems*) [22], in which transitions between states carry both action and probabilistic information. A PTS $M = \langle S, A, T, L, s^i \rangle$ comprises a set of states S, a set of actions A, a transition relation $T \subseteq S \times A \times S$, a probability labeling $L : T \to (0,1]$, and an initial state $s^i \in S$. A transition $(s_1, a, s_2) \in T$ (also written $s_1 \overset{a}{\to} s_2$) indicates that the system can move from state s_1 to state s_2 by performing action a with probability $L(s_1, a, s_2)$. For each state $s \in S$, the probability sum $\sum_{s \overset{a}{\to} s'} L(s, a, s') = 1$.

A path $\sigma = s (= s_0) \overset{a_0}{\to} s_1 \overset{a_1}{\to} \cdots \overset{a_{n-1}}{\to} s_n \cdots$ going out of a state s is an infinite sequence of transitions in M. The i-th state and i-th action of a path σ are noted $\sigma[i]$ and $\sigma_a[i]$, respectively. An interval $\sigma[i, j]$ with $0 \le i \le j$ is the subsequence $\sigma[i] \overset{a_i}{\to} \cdots \overset{a_{j-1}}{\to} \sigma[j]$, which is empty if $i = j$. The suffix starting at the i-th state of a path σ is noted σ_i. The set of paths going out from s is noted $paths_M(s)$. The probability measure of a set of paths sharing a common prefix is defined as $\mu_M(\{\sigma \in paths_M(s) \mid \sigma[0, n] = s_0 \overset{a_0}{\to} \cdots \overset{a_{n-1}}{\to} s_n\}) = L(s_0, a_0, s_1) \times \cdots \times L(s_{n-1}, a_{n-1}, s_n)$.

The regular probabilistic operator that we propose computes the probability measure of paths characterized by regular formulas. For the dataless version of the operator, we use the regular formulas of PDL (*Propositional Dynamic Logic*) [14], defined over the action formulas of ACTL (*Action-based CTL*) [29]. Figure 1 shows the syntax and semantics of the operators.

Action formulas α are built over the set of actions by using standard Boolean connectors. Derived action operators can be defined as usual: $\mathsf{true} = \neg\mathsf{false}$, $\alpha_1 \wedge \alpha_2 = \neg(\neg\alpha_1 \vee \neg\alpha_2)$, etc. Regular formulas β are built from action formulas by using the testing (?), concatenation (.), choice (|), and transitive reflexive closure ($*$) operators. Derived regular operators can be defined as usual: $\mathsf{nil} = \mathsf{false}^*$ is the empty sequence operator, $\beta^+ = \beta.\beta^*$ is the transitive closure operator, etc. State formulas φ are built from Boolean connectors, the possibility modality ($\langle\ \rangle$) and the probabilistic operators ($\{\ \}_{\ge p}$ and $\{\ \}_{>p}$) containing regular formulas. Derived state operators can be defined as usual: $\mathsf{true} = \neg\mathsf{false}$, $\varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2)$, and $[\beta]\,\varphi = \neg\,\langle\beta\rangle\,\neg\varphi$ is the necessity modality.

Action formulas are interpreted on the set of actions A in the usual way. A path satisfies a regular formula β if it has a prefix belonging to the regular language defined by β. The testing operator specifies state formulas that must hold in the intermediate states of a path. Boolean connectors on states are defined as usual. A state s satisfies the possibility modality $\langle\beta\rangle\,\varphi_1$ (resp. the necessity modality $[\beta]\,\varphi_1$) iff some (resp. all) of the paths in $paths_M(s)$ have a prefix satisfying β and leading to a state satisfying φ_1. A state s satisfies the probabilistic operator $\{\beta\}_{\ge p}$ iff the probability measure of the paths in $paths_M(s)$ with a prefix satisfying β is greater or equal to p (and similarly for the strict version of the operator). A PTS $M = \langle S, A, T, L, s^i \rangle$ satisfies a formula φ, denoted by $M \models \varphi$, iff $s^i \models_M \varphi$ (the subscript $_M$ will be omitted when it is clear from the context).

Action formulas:		
$\alpha ::= a$	$b \models_A a$	iff $b = a$
$\mid\ \mathsf{false}$	$b \models_A \mathsf{false}$	iff false
$\mid\ \neg\alpha_1$	$b \models_A \neg\alpha_1$	iff $b \not\models_M \alpha_1$
$\mid\ \alpha_1 \vee \alpha_2$	$b \models_A \alpha_1 \vee \alpha_2$	iff $b \models_M \alpha_1$ or $b \models_M \alpha_2$
Regular formulas:		
$\beta ::= \alpha$	$\sigma[i,j] \models_M \alpha$	iff $i + 1 = j$ and $\sigma_a[i] \models_A \alpha$
$\mid\ \varphi?$	$\sigma[i,j] \models_M \varphi?$	iff $\sigma[i] \models_M \varphi$
$\mid\ \beta_1.\beta_2$	$\sigma[i,j] \models_M \beta_1.\beta_2$	iff $\exists k \in [i,j].\sigma[i,k] \models_M \beta_1$ and $\sigma[k,j] \models_M \beta_2$
$\mid\ \beta_1 \vert \beta_2$	$\sigma[i,j] \models_M \beta_1 \vert \beta_2$	iff $\sigma[i,j] \models_M \beta_1$ or $\sigma[i,j] \models_M \beta_2$
$\mid\ \beta_1^*$	$\sigma[i,j] \models_M \beta_1^*$	iff $\exists k \geq 0.\sigma[i,j] \models_M \beta_1^k$
State formulas:		
$\varphi ::= \mathsf{false}$	$s \models_M \mathsf{false}$	iff false
$\mid\ \neg\varphi_1$	$s \models_M \neg\varphi_1$	iff $s \not\models_M \varphi_1$
$\mid\ \varphi_1 \vee \varphi_2$	$s \models_M \varphi_1 \vee \varphi_2$	iff $s \models_M \varphi_1$ or $s \models_M \varphi_2$
$\mid\ \langle\beta\rangle\, \varphi_1$	$s \models_M \langle\beta\rangle\, \varphi_1$	iff $\exists \sigma \in paths_M(s).\exists i \geq 0.$ $\sigma[0,i] \models_M \beta$ and $\sigma[i] \models_M \varphi$
$\mid\ \{\beta\}_{\geq p}$	$s \models_M \{\beta\}_{\geq p}$	iff $\mu_M(\{\sigma \in paths_M(s) \mid \sigma \models_M \beta\}) \geq p$
$\mid\ \{\beta\}_{> p}$	$s \models_M \{\beta\}_{> p}$	iff $\mu_M(\{\sigma \in paths_M(s) \mid \sigma \models_M \beta\}) > p$

Fig. 1: Modal and probabilistic operators over regular paths

The operator $\{\beta\}_{\geq p}$ generalizes naturally the Until operators of classical probabilistic branching-time logics. The Until operator of PCTL [17], and probabilistic versions of the two Until operators of ACTL are expressed as follows:

$$\begin{aligned} [\varphi_1 \ \mathsf{U}\ \varphi_2]_{\geq p} &= \{(\varphi_1?.\mathsf{true})^*.\varphi_2?\}_{\geq p} \\ \left[\varphi_{1\,\alpha_1} \ \mathsf{U}\ \varphi_2\right]_{\geq p} &= \{(\varphi_1?.\alpha_1)^*.\varphi_2?\}_{\geq p} \\ \left[\varphi_{1\,\alpha_1} \ \mathsf{U}_{\alpha_2}\ \varphi_2\right]_{\geq p} &= \{(\varphi_1?.\alpha_1)^*.\varphi_2?.\alpha_2.\varphi_2?\}_{\geq p} \end{aligned}$$

In addition, regular formulas are strictly more expressive than Until operators, enabling to specify more complex paths in the PTS. For example, the formula:

$$\Psi_1 = \{send.(\mathsf{true}^*.retry)^*.recv\}_{\geq 0.9}$$

unexpressible in (P)ACTL due to the nested $*$-operators, specifies that the probability of receiving a message after zero or more retransmissions is at least 90%.

3 Model Checking Method

We propose below a method for checking a regular probabilistic formula on a PTS on the fly, by reformulating the problem as the simultaneous resolution of a linear equation system (LES) and a Boolean equation system (BES). The method consists of five steps, each one translating the problem into an increasingly concrete intermediate formalism. The first four steps operate syntactically on formulas and their intermediate representations, whereas the fifth step makes use of semantic information contained in the PTS. A detailed formalization of the first four steps, in a state-based setting, can be found in [25]. We illustrate the method by checking the formula Ψ_1 on the PTS of a very simple communication protocol adapted from [3, Chap. 10], shown in Figure 2.

1. Translation to PDL with recursion. To evaluate an operator $\{\beta\}_{\geq p}$ on a PTS $M = \langle S, A, T, L, s^i \rangle$ on the fly, one needs to determine the set of paths going out of s^i and satisfying β, to compute the probability measure of this set, and to compare it with p. For this purpose, it is more appropriate to use an equational representation of β, namely PDLR (*PDL with recursion*), which already served for model checking PDL formulas in the non-probabilistic setting [27]. A PDLR specification is a system of fixed point equations having propositional variables $X \in \mathcal{X}$ in their left hand side and PDL formulas φ in their right hand side:

$$\{X_i = \varphi_i\}_{1 \leq i \leq n}$$

where φ_i are modal state formulas (see Fig. 1) and X_1 is the *variable of interest* corresponding to the desired property. Since formulas φ_i may be open (i.e., contain occurrences of variables X_j), their interpretation is defined w.r.t. a propositional context $\delta : \mathcal{X} \to 2^S$, which assigns state sets to all variables occurring in φ_i. The interpretation of a PDLR specification is the value of X_1 in the least fixed point $\mu\Phi$ of the functional $\Phi : (2^S)^n \to (2^S)^n$ defined by:

$$\Phi(U_1, ..., U_n) = \langle [\![\varphi_i]\!]\, \delta[U_1/X_1, ..., U_n/X_n] \rangle_{1 \leq i \leq n}$$

where $[\![\varphi_i]\!]\, \delta = \{s \in S \mid s \models_\delta \varphi_i\}$, and the interpretation of φ_i (see Fig. 1) is extended with the rule $s \models_\delta X = s \in \delta(X)$. The notation $\delta[U_1/X_1, ..., U_n/X_n]$ stands for the context δ in which X_i were replaced by U_i.

In the sequel, we consider PDLR specifications in *de**r**ivative **n**ormal **f**orm* (RNF), which are the modal logic counterparts of Brzozowski's (generalized) derivatives of regular expressions [5]:

$$\{X_i = \textstyle\bigvee_{j=1}^{n_i} (\varphi_{ij} \wedge \langle \beta_{ij} \rangle\, X_{ij}) \vee \varphi_i\}_{1 \leq i \leq n}$$

where φ_{ij} and φ_i are closed state formulas. Note that, in the right hand side of equation i, the same variable $X_{ij} \in \{X_1, ..., X_n\}$ may occur several times in the first disjunct. Intuitively, a variable X_i denotes the set of states from which there exists a path with a prefix satisfying some of the regular formulas β_{ij} and whose last state satisfies X_{ij}. This is formalized using *path predicates* $P_i : paths_M \to \mathsf{bool}$, defined by the following system of equations:

$$\{P_i(\sigma) = \textstyle\bigvee_{j=1}^{n_i} \exists l_{ij} \geq 0. (\sigma[0] \models \varphi_{ij} \wedge \sigma[0, l_{ij}] \models \beta_{ij} \wedge P_{ij}(\sigma_{l_{ij}})) \vee \sigma[0] \models \varphi_i\}_{1 \leq i \leq n}$$

More precisely, $(\mu\Phi)_i = \{s \in S \mid \exists \sigma \in paths_M(s). P_i(\sigma)\}$.

The PDLR specification in RNF associated to a formula β is defined below:

$$\{X_1 = \langle \beta \rangle\, X_2 \qquad X_2 = \mathsf{true}\}$$

in which the variable of interest X_1 denotes the PDL formula $\langle\beta\rangle\,\mathsf{true}$, expressing the existence of a path with a prefix satisfying β and leading to some final state denoted by X_2. The corresponding path predicates are:

$$\{P_1(\sigma) = \exists l \geq 0.(\sigma[0,l] \models \beta \wedge P_2(\sigma_l)) \qquad P_2(\sigma) = \mathsf{true}\}$$

According to the interpretation of regular formulas (see Fig. 1), the path predicate $P_1(\sigma)$ holds iff $\sigma \models \beta$, and also $(\mu\Phi)_1 = \{s \in S \mid \exists \sigma \in \mathit{paths}_M(s).\sigma \models \beta\}$.
2. Translation to HML with recursion. To bring the PDLR specification closer to an equation system suitable for verification, one must simplify it by removing the regular operators occurring in modalities. This yields a HMLR (*HML with recursion*) specification [21], which contains only HML modalities on action formulas. Regular operators can be eliminated by applying the following substitutions, which are valid equalities in PDL [14]:

$$\begin{array}{rll}
\langle\varphi?\rangle\,X = & \varphi \wedge \langle\mathsf{nil}\rangle\,X & \\
\langle\beta_1.\beta_2\rangle\,X = & \langle\beta_1\rangle\,X' & \text{where } X' = \langle\beta_2\rangle\,X \\
\langle\beta_1|\beta_2\rangle\,X = & \langle\beta_1\rangle\,X \vee \langle\beta_2\rangle\,X & \\
\langle\beta^*\rangle\,X = & \langle\mathsf{nil}\rangle\,X' & \text{where } X' = \langle\mathsf{nil}\rangle\,X \vee \langle\beta\rangle\,X'
\end{array}$$

The rules for the '.' and '*' operators create new equations, necessary for maintaining the PDLR specification in RNF (the insertion of $\langle\mathsf{nil}\rangle\,X$ modalities, which are equivalent to X, serves the same purpose). The rule for the '|' operator creates two occurrences of the same variable X, reflecting that a same state can be reached by two different paths. These rules preserve the path predicates P_i associated to the PDLR specification, and in particular $P_1(\sigma)$, which specifies that a path σ satisfies the initial formula β.

The size of the resulting HMLR specification (number of variables and operators) is linear w.r.t. the size of β (number of operators and action formulas). Besides pure HML modalities, the HMLR specification may also contain occurrences of $\langle\mathsf{nil}\rangle\,X$ modalities, which will be eliminated in the next step.
3. Transformation to guarded form. The right hand side of an equation i of the HMLR specification may contain modalities of the form $\langle\alpha_{ij}\rangle\,Y_{ij}$ and $\langle\mathsf{nil}\rangle\,Y_{ij}$ (equivalent to Y_{ij}), which correspond to *guarded* and *unguarded* occurrences of variables Y_{ij}, respectively. To facilitate the formulation of the verification problem in terms of equation systems, it is useful to remove unguarded occurrences of variables. The general procedure for transforming arbitrary μ-calculus formulas to guarded form [18] can be specialized for HMLR specifications by applying the following actions for each equation i:

1. Remove the unguarded occurrences of X_i in the right hand side of the equation by replacing them with false, which amounts to apply the μ-calculus equality $\mu X.(X \vee \varphi) = \mu X.\varphi$.
2. Substitute all unguarded occurrences of X_i in other equations with the right hand side formula of equation i, and rearrange the right hand sides to maintain the equations in RNF.

This produces a guarded HMLR specification:

$$\{X_i = \bigvee_{j=1}^{n_i} (\varphi_{ij} \wedge \langle \alpha_{ij} \rangle \, X_{ij}) \vee \varphi_i\}_{1 \leq i \leq n}$$

which is the exact modal logic counterpart of Brzozowski's derivatives of regular expressions [5] defined on the alphabet of action formulas. The transformation to guarded form keeps the same number of equations in the HMLR specification, but may increase the number of operators in the right hand sides.

4. Determinization. A HMLR specification may contain, in the right hand side of an equation i, several modalities $\langle \alpha_{ij} \rangle \, X_{ij}$ whose action formulas are not disjoint, i.e., they can match the same action. This denotes a form of nondeterminism, meaning that the same transition $s \xrightarrow{a} s'$ can start a path σ satisfying the path predicate $P_i(\sigma)$ in several ways, corresponding to alternative suffixes of the initial regular formula β. To ensure a correct translation of the verification problem into a LES, it is necessary to determinize the HMLR specification. This can be done by applying the classical subset construction, yielding a deterministic HMLR specification defined on sets of propositional variables:

$$\{X_I = \bigvee_{\emptyset \subset J \subseteq alt(I)} ((\bigwedge_{k \in J} \varphi_k) \wedge \langle \bigwedge_{k \in J} \alpha_k \wedge \bigwedge_{l \in alt(I) \setminus J} \neg \alpha_l \rangle X_J) \vee \bigvee_{i \in I} \varphi_i\}_{I \subseteq [1,n]}$$

where $alt(I) = \{ij \mid i \in I \wedge j \in [1, n_i]\}$. Basically, each alternative $\varphi_{ij} \wedge \langle \alpha_{ij} \rangle \, X_{ij}$ in an equation $i \in I$ is combined with each alternative in the other equations having their index in I, taking care that the action formulas in the resulting modalities are mutually exclusive. As shown in [25] for a similar construction in the state-based setting, the determinization preserves the path predicate associated to the variables of interest X_1 and $X_{\{1\}}$ in the HMLR before and after determinization, i.e., $P_1(\sigma) = P_{\{1\}}(\sigma)$ for any path $\sigma \in paths_M$.

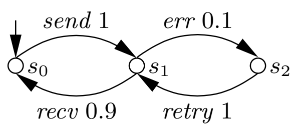
In the worst case, determinization may yield an exponential increase in the size of the HMLR specification. However, this happens on pathological examples of regular formulas, which rarely occur in practice; most of the time, the nondeterminism contained in a formula β is caused by a lack of precision regarding the iteration operators, which can be easily corrected by constraining the action formulas corresponding to iteration "exits". For example, the regular formula contained in Ψ_1 can be made deterministic by specifying precisely the retries and the fact that they must occur before receptions: $send.((\neg retry \wedge \neg recv)^*.retry)^*.recv$. The guarded HMLR specification generated from this latter formula coincides with the determinized one shown in Figure 2. In practice, the number of modalities contained in the determinized HMLR specification can be drastically reduced by determining contradictory combinations and absorptions of action formulas ($\alpha_1 \wedge \alpha_2 = \mathsf{false}$ and $\alpha_1 \wedge \neg \alpha_2 = \alpha_1$ when α_1, α_2 are disjoint).

5. Translation to linear and Boolean equation systems. Consider a determinized HMLR specification in RNF corresponding to a regular formula β:

$$\{X_i = \bigvee_{j=1}^{n_i} (\varphi_{ij} \wedge \langle \alpha_{ij} \rangle \, X_{ij}) \vee \varphi_i\}_{1 \leq i \leq n} \tag{1}$$

where $\alpha_{ij} \wedge \alpha_{ik} = \mathsf{false}$ for each $i \in [1, n]$ and $j, k \in [1, n_i]$. The associated path predicates are defined as follows:

$$\{P_i(\sigma) = \bigvee_{j=1}^{n_i} (\sigma[0] \models \varphi_{ij} \wedge \sigma_a[0] \models \alpha_{ij} \wedge P_{ij}(\sigma_1)) \vee \sigma[0] \models \varphi_i\}_{1 \leq i \leq n} \tag{2}$$



(PTS)

$$\left\{\begin{array}{l} X_1 = \langle send.(\mathsf{true}^*.retry)^*.recv\rangle\, X_2 \\ X_2 = \mathsf{true} \end{array}\right\}$$

(PDLR)

$$\left\{\begin{array}{l} X_1 = \langle send\rangle\, X_3 \\ X_2 = \mathsf{true} \\ X_3 = \langle recv\rangle\, X_2 \vee \langle retry\rangle\, X_3 \vee \langle \mathsf{true}\rangle\, X_6 \\ X_4 = \langle recv\rangle\, X_2 \\ X_5 = \langle retry\rangle\, X_3 \\ X_6 = \langle retry\rangle\, X_3 \vee \langle \mathsf{true}\rangle\, X_6 \end{array}\right\}$$

(guarded HMLR)

$$\left\{\begin{array}{l} X_1 = \langle send\rangle\, X_3 \\ X_2 = \mathsf{true} \\ X_3 = \langle \mathsf{nil}\rangle\, X_4 \vee \langle \mathsf{nil}\rangle\, X_6 \\ X_4 = \langle recv\rangle\, X_2 \\ X_5 = \langle retry\rangle\, X_3 \\ X_6 = \langle \mathsf{nil}\rangle\, X_5 \vee \langle \mathsf{true}\rangle\, X_6 \end{array}\right\}$$

(HMLR)

$$\left\{\begin{array}{l} X_1 = \langle send\rangle\, X_3 \\ X_2 = \mathsf{true} \\ X_3 = \langle recv\rangle\, X_2 \vee \langle retry\rangle\, X_3 \vee \\ \qquad \langle \neg recv \wedge \neg retry\rangle\, X_6 \\ X_6 = \langle retry\rangle\, X_3 \vee \langle \neg retry\rangle\, X_6 \end{array}\right\}$$

(deterministic HMLR)

$$\left\{\begin{array}{l} X_{1,0} = 1.0 \cdot X_{3,1} \\ X_{2,0} = 1.0 \\ X_{3,1} = 0.9 \cdot X_{2,0} + 0.1 \cdot X_{6,2} \\ X_{6,2} = 1.0 \cdot X_{3,1} \end{array}\right\}$$

(LES)

Fig. 2: Model checking formula ψ_1 on a PTS

They are related to the HMLR specification by $(\mu\Phi)_i = \{s \in S \mid \exists \sigma \in paths_M(s).P_i(\sigma)\}$, and to the initial regular formula β by $P_1(\sigma) = \sigma \models \beta$.

The last step of the model checking method reformulates the problem of verifying the determinized HMLR specification on a PTS in terms of solving a LES $(*)$ and a BES $(**)$ defined as follows:

$$\begin{array}{l} \mathcal{P}_i(s) = \boxed{\text{if } s \not\models X_i \text{ then } 0} \\ \qquad \text{else if } s \models \varphi_i \text{ then } 1 \\ \qquad\quad \text{else } \sum_{j=1}^{n_i} \text{if } s \not\models \varphi_{ij} \text{ then } 0 \\ \qquad\qquad\quad \text{else } \sum_{s \xrightarrow{a} s', a \models \alpha_{ij}} L(s,a,s') \times \mathcal{P}_{ij}(s') \end{array} \qquad (*)$$

$$X_i^s = \bigvee_{j=1}^{n_i} (s \models \varphi_{ij} \wedge \bigvee_{s \xrightarrow{a} s'} (a \models \alpha_{ij} \wedge X_{ij}^{s'})) \vee s \models \varphi_i \qquad (**)$$

The LES $(*)$ is obtained by a translation similar to the classical one defined originally for PCTL [17]. A numerical variable $\mathcal{P}_i(s)$ denotes the probability measure of the paths going out of state s and satisfying the path predicate P_i. Determinization guarantees that the sum of coefficients in the right-hand side of each equation is at most 1. The BES $(**)$ is produced by the classical translation employed for model checking modal μ-calculus formulas on LTSs [10, 2]. A Boolean variable X_i^s is true iff state s satisfies the propositional variable X_i of the HMLR specification. The on-the-fly model checking consists in solving the variable $\mathcal{P}_1(s^i)$, which denotes the probability measure of the set of paths going out of the initial state s^i of the PTS and satisfying the initial regular formula β. This is carried out using local LES and BES resolution algorithms, as will be explained in Section 5. The conditions $s \models X_i$ occurring in the LES $(*)$ and the conditions $s \models \varphi_{ij}, s \models \varphi_i$ occurring in both equation systems are checked by applying the on-the-fly model checking method for solving the variable X_i^s of the BES $(**)$ and evaluating the closed state formulas φ_{ij}, φ_i on state s.

4 Extension with Data Handling

The regular formulas that we used so far belong to the dataless fragment [27] of MCL, which considers actions simply as names of communication channels. In practice, the analysis of value-passing concurrent systems, whose actions typically consist of channel names and data values, requires the ability to extract and manipulate these elements. For this purpose, MCL [28] provides action predicates extracting and/or matching data values, regular formulas involving data variables, and parameterized fixed point operators. The regular probabilistic operator $\{\beta\}_{\geq p}$ can be naturally extended with the data handling regular formulas of MCL, which enable to characterize complex paths in a PTS modeling a value-passing concurrent system.

To improve versatility, we extend the regular formulas of MCL with a general iteration operator "loop", which subsumes the classical regular operators with counters, and can also specify paths having a certain cost calculated from the data values carried by its actions. After briefly recalling the main data handling operators of MCL, we define below the "loop" operator, illustrate its expressiveness, and show how the on-the-fly model checking procedure previously described is generalized to deal with the data handling probabilistic operator.

4.1 Overview of data handling MCL operators

In the PTSs modeling value-passing systems, actions are of the form "$C\ v_1 \ldots v_n$", where C is a channel name and $v_1, ..., v_n$ are the data values exchanged during the rendezvous on C. To handle the data contained in actions, MCL provides *action predicates* of the form "$\{C\ ...\ !e\ ?x{:}T$ where $b\}$", where "..." is a wildcard matching zero or more data values of an action, e is an expression whose value matches the corresponding data value, x is a data variable of type T that is initialized with the corresponding data value extracted from the action, and b is an optional boolean expression (guard) typically expressing a condition on x. An action predicate may contain several clauses "$!e$" and "$?x{:}T$", all variables defined by "$?x{:}T$" clauses being visible in the guard b and also outside the action predicate. An action satisfies an action predicate if its structure is compatible with the clauses of the predicate, and the guard evaluates to true in the context of the data variables extracted from the action.

Regular formulas in MCL are built over action predicates using the classical operators shown in Section 2, as well as constructs inspired from sequential programming languages: conditional ("if-then-else"), counting, iteration ("for" and "loop", described in the next subsection), and definition of variables ("let"). The testing operator of PDL is expressed in MCL as $\varphi? = \textsf{if } \neg\varphi \textsf{ then false end if}$.

Finally, the state formulas of MCL are built using modalities containing regular formulas, parameterized fixed point operators, quantifiers over finite domains, and programming language constructs ("if" and "let").

4.2 Generalized iteration on regular formulas

The general iteration mechanism that we propose on regular formulas consists of three operators having the following syntax:

$$\beta \ ::= \ \textsf{loop } (x{:}T{:=}e_0) \textsf{ : } (x'{:}T') \textsf{ in } \beta \textsf{ end loop} \ \mid \ \textsf{continue } (e) \ \mid \ \textsf{exit } (e')$$

The "loop" operator denotes a path made by concatenation of (zero or more) path fragments satisfying β, each one corresponding to an iteration of the loop with the current value of variable x. Variable x, which is visible inside β, is initialized with the value of expression e_0 at the first loop iteration and can be updated to the value of e by using the operator "continue (e)", which starts a new iteration of the loop. The loop is terminated by means of the "exit (e')" operator, which sets the return variable x', visible outside the "loop" formula, to the value of e'.

The iteration and return variables (x and x') are both optional; if they are absent, the "in" keyword is also omitted. For simplicity, we used only one variable x and x', but several variables of each kind are allowed. The arguments of the operators "continue" and "exit" invoked in the loop body β must be compatible with the declarations of iteration and return variables, respectively. Every occurrence of "continue" and "exit" refers to the immediately enclosing "loop", which enforces a specification style similar to structured programming.

For brevity, we define the semantics of the "loop" operator by translating it to plain MCL in the context of an enclosing diamond modality. The translation is parameterized by a profile $Z/x{:}T/x'{:}T'$, where x and x' are the iteration and return data variables of the immediately enclosing "loop", and Z is a propositional variable associated to it. We show below the translation of the three general iteration operators, the other regular operators being left unchanged.

$$\left(\left\langle \begin{array}{l} \textsf{loop } (x{:}T{:=}e_0) \textsf{ : } (x'{:}T') \textsf{ in} \\ \quad \beta \\ \textsf{end loop} \end{array} \right\rangle \varphi \right)_{Z/x:T/x':T'} \overset{\text{def}}{=} \mu W(x{:}T{:=}e_0).\, \langle (\beta)_{W/x:T/x':T'} \rangle\, \varphi$$

$$(\langle \textsf{continue } (e) \rangle\, \varphi)_{Z/x:T/x':T'} \overset{\text{def}}{=} Z(e)$$

$$(\langle \textsf{exit } (e') \rangle\, \varphi)_{Z/x:T/x':T'} \overset{\text{def}}{=} \textsf{let } x'{:}T' := e' \textsf{ in } \varphi \textsf{ end let}$$

Basically, a possibility modality enclosing a "loop" operator is translated into a minimal fixed point operator parameterized by the iteration variable(s). The occurrences of "continue" in the body of the loop are translated into invocations of the propositional variable with the corresponding arguments, and the occurrences of "exit" are translated into "let" state formulas defining the return variables and setting them to the corresponding return values.

Syntax	Meaning	Encoding using "loop"
β^*	≥ 0 times	loop exit | β . continue end loop
β^+	≥ 1 times	loop β . (exit | continue) end loop

$\beta\{e_1 \ldots e_2\}$	between e_1 and e_2 times	loop (c_1:nat := e_1, c_2:nat := $e_2 - e_1$) in
		  if $c_1 > 0$ then β . continue ($c_1 - 1, c_2$)
		  elsif $c_2 > 0$ then exit | β . continue ($c_1, c_2 - 1$)
		  else exit end if
		end loop

for n:nat from e_1 to e_2
  step e_3 do
  β
end for
	stepwise	loop (n:nat := e_1) in
		  if $n < e_2$ then β . continue ($n + e_3$)
		  else exit end if
		end loop

All iteration operators on MCL regular formulas can be expressed in terms of the "loop" operator, as shown in the table above. For simplicity, we omitted the definitions of $\beta\{e\}$ (iteration e times) and $\beta\{\ldots e\}$ (iteration at most e times), which are equivalent to $\beta\{e \ldots e\}$ and $\beta\{0 \ldots e\}$, respectively. To illustrate the semantics of general iteration, consider the formula $\langle\beta\{e\}\rangle$ true stating the existence of a path made of e path fragments satisfying β. By applying the encoding of bounded iteration in terms of "loop" and the translation rules of general iteration, we obtain:

$$\langle\beta\{e\}\rangle\,\mathsf{true} \;=\; \left\langle \begin{array}{l} \mathsf{loop}\ (c{:}\mathsf{nat} := e)\ \mathsf{in} \\ \quad \mathsf{if}\ c > 0\ \mathsf{then} \\ \quad\quad \beta\ .\ \mathsf{continue}\ (c-1) \\ \quad \mathsf{else\ exit\ end\ if} \\ \mathsf{end\ loop} \end{array} \right\rangle \mathsf{true} \;=\; \begin{array}{l} \mu Z(c{:}\mathsf{nat} := e). \\ \quad \mathsf{if}\ c > 0\ \mathsf{then}\ \langle\beta\rangle\, Z(c-1) \\ \quad \mathsf{else} \\ \quad\quad \mathsf{true} \\ \quad \mathsf{end\ if} \end{array}$$

The bounded iteration operators $\beta\{e\}$, $\beta\{\ldots e\}$, and $\beta\{e_1 \ldots e_2\}$ are natural means for counting actions (ticks), and hence describing discrete-time properties. The full Until operator of PCTL, and its action-based counterparts derived from ACTL, can be expressed as follows ($t \geq 0$ is the number of ticks until φ_2):

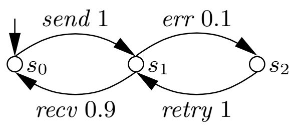
$$\begin{aligned} [\varphi_1\ \mathsf{U}\ \varphi_2]^{\leq t}_{\geq p} &= \{(\varphi_1?.\mathsf{true})\{0 \ldots t\}.\varphi_2?\}_{\geq p} \\ [\varphi_{1\,\alpha_1}\ \mathsf{U}\ \varphi_2]^{\leq t}_{\geq p} &= \{(\varphi_1?.\alpha_1)\{0 \ldots t\}.\varphi_2?\}_{\geq p} \\ [\varphi_{1\,\alpha_1}\ \mathsf{U}_{\alpha_2}\ \varphi_2]^{\leq t}_{\geq p} &= \{(\varphi_1?.\alpha_1)\{0 \ldots t\}.\varphi_2?.\alpha_2.\varphi_2?\}_{\geq p} \end{aligned}$$

Besides counting, the general iteration operators are able to characterize complex paths in a PTS, by collecting the data values (costs) present on actions and using them in arbitrary computations (see the examples in Section 5).

4.3 Model checking method with data handling

The on-the-fly model checking method shown in Section 3 can be generalized to deal with the data handling constructs of MCL by adding data parameters to the various equation systems used as intermediate forms. We illustrate the complete method by checking the formula Ψ_2 on the PTS shown on Figure 2:

$$\Psi_2 = \{send.((\neg retry \wedge \neg recv)^*.retry)\{\ldots n\}.recv\}_{\geq 0.9}$$



(a) PTS

$$\left\{\begin{array}{l} X_1 = \langle \mathit{send}.((\neg \mathit{retry} \wedge \neg \mathit{recv})^*. \\ \qquad\qquad \mathit{retry})\{\ldots\ n\}.\mathit{recv}\rangle X_2 \\ X_2 = \mathsf{true} \end{array}\right\}$$

(b) PDLR

$$\left\{\begin{array}{l} X_1 = \langle \mathit{send}\rangle\, X_3 \\ X_2 = \mathsf{true} \\ X_3 = \langle((\neg \mathit{retry} \wedge \neg \mathit{recv})^*. \\ \qquad\quad \mathit{retry})\{\ldots\ n\}\rangle X_4 \\ X_4 = \langle \mathit{recv}\rangle\, X_2 \end{array}\right\}$$

(c) PDLR

$$\left\{\begin{array}{l} X_1 = \langle \mathit{send}\rangle\, X_3 \\ X_2 = \mathsf{true} \\ X_3 = \langle \mathsf{loop}\ (c{:}\mathsf{nat}{:=}n)\ \mathsf{in} \\ \qquad\quad \mathsf{if}\ c > 0\ \mathsf{then} \\ \qquad\qquad \mathsf{exit}\ | \\ \qquad\qquad (\neg \mathit{retry} \wedge \neg \mathit{recv})^*. \\ \qquad\qquad \mathit{retry}.\mathsf{continue}\ (c-1) \\ \qquad\quad \mathsf{else\ exit\ end\ if} \\ \qquad \mathsf{end\ loop}\rangle X_4 \\ X_4 = \langle \mathit{recv}\rangle\, X_2 \end{array}\right\}$$

(d) PDLR

$$\left\{\begin{array}{rl} X_1 &= \langle \mathit{send}\rangle\, X_3 \\ X_2 &= \mathsf{true} \\ X_3 &= \langle \mathsf{nil}\rangle\, X_5(n) \\ X_4 &= \langle \mathit{recv}\rangle\, X_2 \\ X_5(c) &= \mathsf{if}\ c > 0\ \mathsf{then} \\ & \qquad\quad \langle \mathsf{nil}\rangle\, X_4 \vee \langle \mathsf{nil}\rangle\, X_6(c-1) \\ & \quad\ \mathsf{else}\ \langle \mathsf{nil}\rangle\, X_4\ \mathsf{end\ if} \\ X_6(c) &= \langle \mathsf{nil}\rangle\, X_8(c) \\ X_7(c) &= \langle \mathit{retry}\rangle\, X_5(c) \\ X_8(c) &= \langle \mathsf{nil}\rangle\, X_7(c) \vee \langle \neg \mathit{retry} \wedge \neg \mathit{recv}\rangle\, X_8(c) \end{array}\right\}$$

(e) HMLR

$$\left\{\begin{array}{rl} X_1 &= \langle \mathit{send}\rangle\, X_5(n) \\ X_2 &= \mathsf{true} \\ X_5(c) &= \mathsf{if}\ c > 0\ \mathsf{then} \\ & \qquad\quad \langle \mathit{recv}\rangle\, X_2 \vee \langle \mathit{retry}\rangle\, X_5(c-1) \vee \\ & \qquad\quad \langle \neg \mathit{retry} \wedge \neg \mathit{recv}\rangle\, X_8(c-1) \\ & \quad\ \mathsf{else}\ \langle \mathit{recv}\rangle\, X_2\ \mathsf{end\ if} \\ X_8(c) &= \langle \mathit{retry}\rangle\, X_5(c) \vee \langle \neg \mathit{retry} \wedge \neg \mathit{recv}\rangle\, X_8(c) \end{array}\right\}$$

(f) guarded HMLR

$$\left\{\begin{array}{rl} X_{1,0} &= 1.0 \cdot X_{5,1}(n) \\ X_{2,0} &= 1.0 \\ X_{5,1}(c) &= \mathsf{if}\ c > 0\ \mathsf{then} \\ & \qquad 0.9 \cdot X_{2,0} + 0.1 \cdot X_{8,2}(c-1) \\ & \quad\ \mathsf{else}\ 0.9 \cdot X_{2,0}\ \mathsf{end\ if} \\ X_{8,2}(c) &= 1.0 \cdot X_{5,1}(c) \end{array}\right\}$$

(g) parameterized LES

$$\left\{\begin{array}{rl} X_{1,0} &= 1.0 \cdot X_{5,1}(2) \\ X_{5,1}(2) &= 0.9 + 0.1 \cdot X_{5,1}(1) \\ X_{5,1}(1) &= 0.9 + 0.1 \cdot X_{5,1}(0) \\ X_{5,1}(0) &= 0.9 \end{array}\right\}$$

(h) LES instantiated for $n = 2$

Fig. 3: Model checking formula ψ_2 on a PTS

Formula Ψ_2, which is a determinized data-based variant of Ψ_1, specifies that the probability of receiving a message after at most n retransmissions (where n is a parameter to be instantiated) is at least 90%.

The various translation phases are illustrated on Figure 3. The translation rules for standard regular operators given in Section 3 are applied for eliminating the "." operators in the PDLR specification (Fig. 3(c)). Then, the iteration at most n times is translated into a "loop" operator (Fig. 3(d)), and the corresponding modality is further refined using the semantics of "loop" defined in Section 4.2, yielding a HMLR specification parameterized by a counter (Fig. 3(e)). After bringing this specification to guarded form (Fig. 3(f)), a parameterized LES is produced by applying the translation scheme given in Section 3 extended to handle data parameters. For instance, variable $X_{5,1}(v)$ in the LES denotes the probability measure of the paths starting from state s_1 and satisfying the path predicate denoted by X_5 with the parameter c set to value v. Finally, a plain LES is generated (Fig. 3(g)) by instantiating $n = 2$ in the parameterized LES. Note that the guarded HMLR specification was already deterministic (since the regular formula in Ψ_2 was determinized), and hence the LES has a unique solution. By solving this LES (e.g., using substitution), we obtain $X_{1,0} = 0.999$, which is the probability measure of the paths starting from the initial state s_0 of the PTS and satisfying the regular formula specified in Ψ_2. In other words, $n = 2$ retransmissions ensure that a message is received with 99.9% probability.
Termination. The presence of data parameters (with infinite domains) implies that the whole model checking procedure relies on the termination of the instantiation phase, which must create a finite LES solvable using numerical methods. This is in general undecidable, similarly to the termination of term rewriting [12]. Such situations happen for "pathological" formulas, which carry on divergent computations on data unrelated to the data values contained in the PTS actions. For example, the modality $\langle \mathsf{loop}\ (k\mathsf{:nat:=}0)\ \mathsf{in}\ a\ .\ \mathsf{continue}\ (k+1)\ \mathsf{end\ loop}\rangle\, \mathsf{true}$ will not converge on the PTS consisting of a single loop $s \xrightarrow{a} s$, since it will entail the construction of an infinite LES $\{X_s(0) = X_s(1), X_s(1) = X_s(2), ...\}$. However, the model checking procedure terminates for most practical cases of data handling regular formulas (counting, accumulating or aggregating values, computing costs over paths).

5 Tool Support and Use

In this section, we show how the on-the-fly model checking method for the regular probabilistic operator works in practice. After briefly presenting the implementation of the method within the CADP toolbox [16], we illustrate its application for the quantitative analysis of shared-memory mutual exclusion protocols.

5.1 Implementation

We extended MCL with the general iteration operator "loop" on regular formulas and the regular probabilistic operator $\{\beta\}_{\bowtie p}$, where $\bowtie \in \{<, \leq, >, \geq, =\}$.

Temporal and probabilistic operators can be freely combined, e.g., $[\beta_1] \{\beta_2\}_{\geq p}$ specifies that, from all states reached after a path satisfying β_1, the probability measure of an outgoing path satisfying β_2 is at least p.

We also enhanced the EVALUATOR [28] on-the-fly model checker with the translation of $\{\beta\}_{\bowtie p}$ formulas into BESs (for checking the existence of path suffixes) and LESs (for computing probability measures) as described in Sections 3 and 4. The on-the-fly resolution of BESs is carried out by the algorithms of the CAESAR_SOLVE library [24], which already serves as verification backend for (non-probabilistic) MCL formulas. For the on-the-fly resolution of LESs, we designed a local algorithm operating on the associated Signal Flow Graphs (SFG) [7], in a way similar to the BES resolution algorithms, which operate on the associated Boolean graphs [2]. The LES resolution algorithm consists of a forward exploration of the SFG to build dependencies between variables, followed by a backward variable elimination (a.k.a. substitution) and a final propagation to update the right-hand sides of equations with the solutions of variables. The number of floating-point operations is reduced by a careful bookkeeping of dependencies between variables. This substitution method performs well on LESs resulting from $\{\beta\}_{\bowtie p}$ operators, which exhibit a high degree of sparsity (typically, for a PTS with a branching factor of 10 and having 10^6 states, there are about $10^{-5} = 0.001\%$ non-null elements in the LES matrix).

5.2 Case study: analysis of mutual exclusion protocols

We illustrate the application of the regular probabilistic operator by carrying out a quantitative analysis of several shared-memory mutual exclusion protocols, using their formal descriptions in LNT [6] given in [26]. We focus here on a subset of the 27 protocols studied in [26], namely the CLH, MCS, Burns&Lynch (BL), TAS and TTAS protocols, by considering configurations of up to $N = 4$ concurrent processes competing to access the critical section. Each process executes cyclically a sequence of four sections: non critical, entry, critical, and exit. The entry and exit sections represent the algorithm specific to each protocol for demanding and releasing the access to the critical section, respectively. In the PTS models of the protocols, all transitions going out from each state are assumed to have equal probabilities. We formulate four probabilistic properties using MCL and evaluate them on the fly on each LNT protocol description. To compute probabilities, we also use the fact that, besides the Boolean verdict, the model checker also yields the value of the probability measure for the MCL formulas consisting of a single $\{\beta\}_{\bowtie p}$ operator. For each property requiring several invocations of the model checker with different values for the data parameters in the MCL formula, we automate the analysis using SVL scripts [15].
Critical section. First of all, for each $i \in [0, N-1]$, we compute the probability that process P_i is the first one to enter its critical section. For this purpose, we use the following MCL formula:

$$\{(\neg\{\mathsf{CS}\ !\text{"ENTER"}\ \ldots\})^*.\{\mathsf{CS}\ !\text{"ENTER"}\ !i\}\}_{\geq 0}$$

which computes the probability that, from the initial state, process P_i accesses its critical section before any (other) process. Symmetric protocols guarantee that this probability is equal to $1/N$ for all processes, while asymmetric protocols (such as BL) may favor certain processes w.r.t. the others. This is indeed reflected by the results of model checking the above formula for $N = 3$: for the BL protocol, which gives higher priority to processes of lower index, the probabilities computed are 72.59% (for P_0), 21.66% (for P_1), and 5.73% (for P_2), whereas they are equal to 33.33% for the other protocols, which are symmetric.

Memory latency. The analysis of critical section reachability can be refined by taking into account the cost of memory accesses (e.g., read, write, test-and-set operations on shared variables) that a process P_i must perform before entering its critical section. The protocol modeling provided in [26] also considers non-uniform memory accesses, assuming that concurrent processes execute on a cache-coherent multiprocessor architecture. The cost c (or latency) of a memory access depends on the placement of the memory in the hierarchy (local caches, shared RAM, remote disks) and is captured in the PTS by means of additional actions "MU !c" [26].

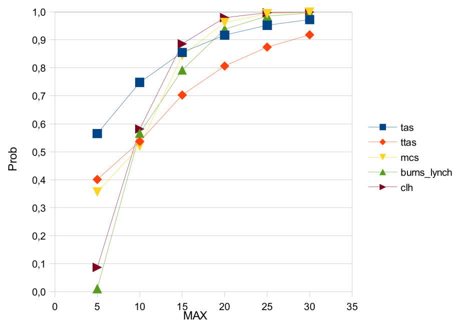
The MCL formula below computes the probability that a process P_i performs memory accesses of a total cost **max** before entering its critical section:

```
{ (¬{NCS !i})*.{NCS !i}.
    loop (total_cost:nat:=0) in
        (¬({MU ... !i} ∨ {CS !"ENTER" !i}))*.
        if total_cost < max then
            {MU ... ?c:nat !i} . continue (total_cost + c)
        else
            exit
        end if
    end loop .
    {CS !"ENTER" !i} }≥0
```

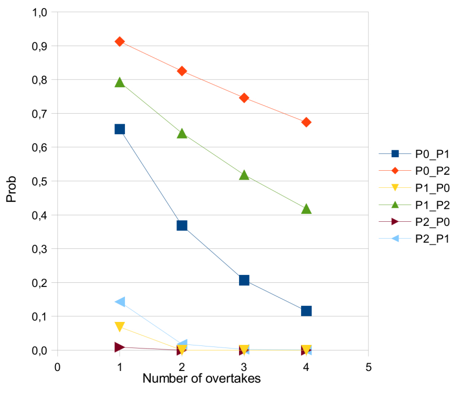
The regular formula above expresses that, after executing its non critical section for the first time, process P_i begins its entry section and, after a number of memory accesses, enters its critical section. The "loop" subformula denotes the entry section of P_i and ensures that this section terminates as soon as the cost of all memory accesses performed by P_i (accumulated in the iteration parameter *total_cost*) exceeds a given value **max**. The other processes can execute freely during the entry section of P_i, in particular (depending on the protocol) they can overtake P_i by accessing their critical sections before it. Figure 4(a) shows the probability of entering the critical section for various values of **max**.

Due to the presence of waiting loops in the entry section, the maximal number of memory accesses of P_i before entering its critical section is unbounded (and hence, also the cost **max**). However, the probability that a process waits indefinitely before entering its critical section tends to zero in long-term runs of starvation-free protocols. This explains why an asymptotic probability of 1.0 is observed in Figure 4(a): a process has better chances to reach its critical section when the memory cost of its entry section increases.

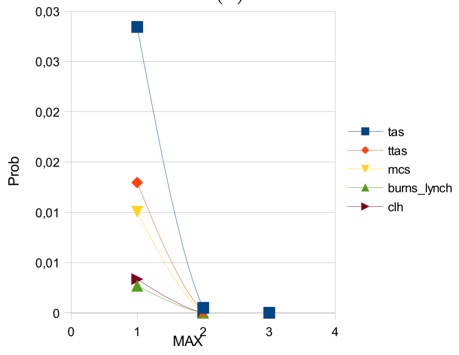
We can further refine the MCL formula above to infer the steady-state behavior by imposing a number of protocol executions before the entry section of P_i (i.e., start accumulating the memory access cost after the r-th “NCS !i” action). For the TAS protocol, we observed a convergence of the probability when $r = 5$ for values of **max** between 5 (steady-state probability of 42.66%) and 30 (steady-state probability of 96.41%). Other properties involving the memory latency can be expressed similarly, e.g., compute the probability to enter the critical section after executing an entry section that maximizes the ratio of local versus remote memory accesses.



(a)



(b)



(c)

Fig. 4: Probabilities computed using on-the-fly model checking. (a) Accessing the critical section after memory accesses of cost MAX. (b) Overtaking of P_i by P_j (P_j_P_i) in the BL protocol. (c) Standalone execution of P_i.

Overtaking. Even if a mutual exclusion protocol is starvation-free, a process P_i that begins its entry section (and hence, starts requesting the access to the critical section) may be overtaken one or several times by another process P_j that accesses its own critical section before P_i does so. A qualitative measure of a starvation-free protocol is given by its *overtaking degree*, which is the maximum number of overtakes per couple of processes. This number should be as small as possible, and may vary among process couples for asymmetric protocols.

A qualitative study of the overtaking degree was carried out in [26] using MCL regular formulas with counters. Here we use the same property in the probabilistic setting, which enables to compute the probability that process P_j overtakes P_i a given number of times. Figure 4(b) shows the results for the BL protocol, which outline its intrinsic asymmetry: lower index processes, with higher priority, also have better chances to overtake the other processes.

Standalone execution. As opposed to overtaking, it is also interesting to examine the dual situation, in which a process P_i executes its cycle in standalone, i.e., without any interference with the other processes. This situation was explicitly formulated in [13] as the *independent progress* requirement, which should be satisfied by any mutual exclusion protocol. We can analyze this situation by computing the probability measure of a complete execution of process P_i without any other action being performed meanwhile by other processes. This execution can be specified using the MCL formula below:

$$\begin{array}{l} \{\ \ ((\neg\{\mathsf{CS}\ ...\ ?j{:}\mathsf{nat\ where}\ j \neq i\})^*.\{\mathsf{NCS}\ !i\}. \\ \quad (\neg\{...\ ?j{:}\mathsf{nat\ where}\ j \neq i\})^*.\{\mathsf{CS}\ !\text{"ENTER"}\ !i\}. \\ \quad (\neg\{...\ ?j{:}\mathsf{nat\ where}\ j \neq i\})^*.\{\mathsf{CS}\ !\text{"LEAVE"}\ !i\}. \\ \quad)\ \{\mathbf{max}\}\ \ \}_{\geq 0} \end{array}$$

where **max** denotes the number of consecutive executions of P_i. Figure 4(c) shows that the probabilities of standalone execution of P_i in a pool of four processes decrease with **max**, which reflects the starvation-free nature of the protocols.

Performance of analysis. All model checking experiments have been carried out in a single core of an Intel(R) Xeon(R) E5-2630v3 @2.4GHz with 128 GBytes of RAM and Linux Debian 7.9 within a cluster of Grid'5000 [4]. The sizes of the PTSs, including the additional transitions representing the memory access costs, range from 3 252 states and 6 444 transitions (for the TAS protocol) to 18 317 849 states and 31 849 616 transitions (for the CLH protocol).

The computing resources needed for on-the-fly verification are variable depending on the complexity of the MCL regular formulas, and in particular the number and domains of their data parameters. For example, the analysis of the first access to the critical section takes between 3.25-5.5 seconds and 36.5-77 MBytes for all protocol configurations considered. For other properties, such as those concerning the memory latency or the overtaking, some peaks arrive up to 2-3 hours and 12-14 GBytes because of the manipulation of data (cost of memory accesses) and iterations (number of overtakes). The analysis of the standalone execution of P_i may take up to 285 seconds and 1 230 MBytes for the BL protocol because of the complex cycles present in the PTS, while the same analysis takes less than 100 seconds (or even 10 seconds) for the other protocols.

6 Conclusion and Future Work

We proposed a regular probabilistic operator for computing the probability measure of complex paths in a PTS whose actions contain data values. Paths are specified using the action-based, data handling regular formulas of MCL [28] that we extended with a general iteration operator "loop" enabling the specification of arbitrarily complex paths. These new operators subsume those of P(A)CTL, and make possible the study of paths whose associated cost (calculated from the data values present on their actions) has a given value. We defined an on-the-fly model checking method based on reformulating the problem as the resolution of a linear equation system (LES) and a Boolean equation system (BES), and implemented it in the EVALUATOR model checker of the CADP toolbox.

Regarding future work, we plan to investigate the adequacy of MCL fragments w.r.t. probabilistic branching bisimulation, which would enable the (property-preserving) compositional construction and minimization of PTSs described as Interactive Probabilistic Chains [11]. We also plan to bring distributed capabilities to the on-the-fly analysis back-end (which is currently sequential), by connecting the model checker with the MUMPS distributed solver [1] for sparse LESs. Finally, we will seek to extend the proposed approach to handle infinite sequences, described in MCL by means of data handling fairness operators similar to generalized Büchi automata.

Acknowledgments This work was supported by the European project SENSATION (*Self Energy-Supporting Autonomous Computation*) FP7-318490.

References

1. P. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. MUMPS: A general purpose distributed memory sparse solver. In *PARA'2000*, LNCS vol. 1947, p. 121–130. Springer, 2000.
2. H. R. Andersen. Model checking and boolean graphs. *TCS*, 126(1):3–30, 1994.
3. C. Baier and J.-P. Katoen. *Principles of model checking.* MIT Press, 2008.
4. R. Bolze, F. Cappello, E. Caron, M. J. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quétier, O. Richard, E.-G. Talbi, and I. Touche. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *IJHPCA*, 20(4):481–494, 2006.
5. J. A. Brzozowski. Derivatives of regular expressions. *JACM*, 11(4):481–494, 1964.
6. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, C. McKinty, V. Powazny, F. Lang, W. Serwe, and G. Smeding. Reference Manual of the LNT to LOTOS Translator (Version 6.2). Inria/Vasy and Inria/Convecs, 130 pages, 2015.
7. L. O. Chua and P. M. Lin. *Computer Aided Analysis of Electronic Circuits.* Prentice Hall, 1975.
8. E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2000.
9. R. Cleaveland, S. P. Iyer, and M. Narasimha. Probabilistic temporal logics via the modal mu-calculus. *TCS*, 342(2-3):316–350, 2005.
10. R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *FMSD*, 2(2):121–147, 1993.
11. N. Coste, H. Hermanns, E. Lantreibecq, and W. Serwe. Towards performance prediction of compositional models in industrial gals designs. In *CAV'2009*, LNCS vol. 5643, p. 204–218. Springer, 2009.
12. N. Dershowitz. Termination of rewriting. *J. of Symb. Comput.*, 3(1):69–115, 1987.
13. E. W. Dijkstra. Solution of a problem in concurrent programming control. *CACM*, 8(9):569, 1965.

14. M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *JCSS*, 18(2):194–211, 1979.
15. H. Garavel and F. Lang. SVL: a Scripting Language for Compositional Verification. In *FORTE'01*, p. 377–392. Kluwer, 2001.
16. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *STTT*, 15(2):89–107, 2013.
17. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Asp. Comput.*, 6(5):512–535, 1994.
18. D. Kozen. Results on the propositional μ-calculus. *TCS*, 27:333–354, 1983.
19. D. Kozen. A probabilistic PDL. *JCSS*, 30(2):162–178, 1985.
20. M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV'2011*, LNCS vol. 6806, p. 585–591. Springer, 2011.
21. K. G. Larsen. Proof systems for hennessy-milner logic with recursion. In *CAAP'88*, LNCS vol. 299, p. 215–230. Springer, 1988.
22. K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Inf. and Comput.*, 94(1):1–28, 1991.
23. D. Latella, M. Loreti, and M. Massink. On-the-fly fast mean-field model-checking. In *Trustworthy Global Computing*, p. 297–314. Springer, 2014.
24. R. Mateescu. Caesar_solve: A generic library for on-the-fly resolution of alternation-free boolean equation systems. *STTT*, 8(1):37–56, 2006.
25. R. Mateescu, P. T. Monteiro, E. Dumas, and H. de Jong. CTRL: Extension of CTL with regular expressions and fairness operators to verify genetic regulatory networks. *TCS*, 412(26):2854–2883, 2011.
26. R. Mateescu and W. Serwe. Model Checking and Performance Evaluation with CADP Illustrated on Shared-Memory Mutual Exclusion Protocols. *SCP*, 78(7):843–861, 2013.
27. R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *SCP*, 46(3):255–281, 2003.
28. R. Mateescu and D. Thivolle. A model checking language for concurrent value-passing systems. In *FM'08*, LNCS vol. 5014, p. 148–164. Springer, 2008.
29. R. De Nicola and F. W. Vaandrager. Action versus State Based Logics for Transition Systems. In *Semantics of concurrency*, LNCS vol. 469, p. 407–419. Springer, 1990.