Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

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

Radu Mateescu[1] **and José Ignacio Requeno**[1,2]

[1] Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
[2] University of Zaragoza, 50009 Zaragoza, Spain

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 specifies the probability measure of a path described 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 [14, 7]. One of the first logics capturing behavioral, discrete time, and probabilistic information is PCTL (*Probabilistic Computation Tree Logic*) [29].

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*) [39], which are suitable action-based models for representing value-passing concurrent systems with interleaving semantics. We consider here the class of *generative* (also called *fully probabilitistic*) PTSs [28], in which transition probability distributions implicitly assign probabilities to the occurrences of actions. These PTSs can be viewed as DTMCs (*Discrete-Time Markov Chains*) in which transitions are labeled by actions carrying probabilities, channel names, and data values sent between concurrent processes during handshake communication. We are interested mainly in action-based properties, in contrast with standard DTMCs [7, Chap. 10], where atomic propositions are attached to states. Our contributions are twofold.

Regarding the specification of properties, we propose a new regular probabilistic operator, which specifies 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*) [39] is a variant of HML (*Hennessy-Milner Logic*) [30] with modalities indexed by probabilities, and was introduced as a modal characterization of probabilistic bisimulation on (generative) PTSs. GPL (*Generalized Probabilistic Logic*) [15] 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*) [24]. To handle the data values present on PTS actions, we rely on the regular formulas with counters of MCL (*Model Checking Language*) [48], 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 complex (Turing computable) 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 in the size of the PTS and linear (resp. exponential) in 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 [40], all previous implementations, e.g., in PRISM [36] 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*) [35], which are subsumed by the regular probabilistic operator of MCL. We implemented the method in the EVALUATOR [48] on-the-fly model checker of the CADP toolbox [27] and experimented it on various examples of value-passing concurrent systems.

Related work. Most of the works on action-based logics for quantitative analysis have concentrated on continuous-time models. CSL (*Continuous Stochastic Logic*) [4,6] is a continuous-time variant of PCTL interpreted on CTMCs (*Continuous-Time Markov Chains*). aCSL (*action-based CSL*) [31], interpreted on action-labeled CTMCs, is the action-based counterpart of CSL suitable for action-oriented modeling formalisms, such as stochastic process algebras. aCSL was defined in a way similar to ACTL (*Action-based CTL*) [49], which is the action-based counterpart of CTL [13]. In our setting, we can encode probabilistic versions of the ACTL operators, which are discrete-time counterparts of the time-bounded probability operators of aCSL. This logic was subsequently extended with regular operators similar to those of PDL, leading to the strictly more expressive logic asCSL [5] interpreted on state- and action-labeled CTMCs. The purpose of asCSL was to enable the specification of complex properties of finite computations with real-time constraints. We had a similar motivation for our regular probabilistic operator on PTSs, and we extended its expressiveness further by adding data-handling mechanisms.

In the state-based setting, the logic closest to ours is QuaTEx (*Quantitative Temporal Expressions*) [1], which combines constructs from PCTL and the rule-based query language EAGLE [8]. In QuaTEx, paths are described using recursive temporal operators with data parameters, built over a next-time operator and conditional path expressions. QuaTEx queries are evaluated using statistical model checking on GSMPs (*Generalized Semi-Markov Processes*) produced from actor-based probabilistic programs written in the PMaude language [1]. Instead of recursive path formulas, our data-handling probabilistic operator extends the regular expressions over paths with a generalized iteration operator, yielding more concise specifications of step-bounded properties.

Organization of the paper. This paper is an extended version of a conference paper [45], to which it adds the following material: (i) an enhanced review of related work; (ii) an in-depth presentation of the proposed logic, with more examples of properties, and of the proposed on-the-fly model checking procedure; (iii) further experimental validation on three probabilistic systems (BRP protocol and two randomized protocols) and a comparison with the PRISM model checker.

The rest of 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 Section 6 illustrates it for the quantitative analysis of several protocols. Finally, Section 7 gives concluding remarks and directions of future work.

2 Dataless regular probabilistic operator

As interpretation models, we consider generative PTSs (*Probabilistic Transition Systems*) [39], in which transitions between states carry both action and probabilistic information. A PTS $M = \langle S, A, T, \mathbf{P}, s^i \rangle$ comprises a set of states S, a set of actions A, a transition relation $T \subseteq S \times A \times S$, a probability labeling $\mathbf{P} : T \to (0, 1]$, and an initial state $s^i \in S$. A transition $(s, a, s') \in T$ (also written $s \xrightarrow{a} s'$) indicates that the system can move from state s to state s' by performing action a with probability $\mathbf{P}(s, a, s')$. For each state $s \in S$, the probability sum $\sum_{s \xrightarrow{a} s'} \mathbf{P}(s, a, s') = 1$. These PTS models do not contain nondeterminism, but probabilistic choice, i.e., a transition $s \xrightarrow{a} s'$ assigns implicitly the same probability $\mathbf{P}(s, a, s')$ to the occurrence of a.

Action formulas:		
$\alpha ::= a$	$b \models_A a$	iff $b = a$
$\mid$ false	$b \models_A$ false	iff false
$\mid \neg\alpha_1$	$b \models_A \neg\alpha_1$	iff $b \not\models_A \alpha_1$
$\mid \alpha_1 \vee \alpha_2$	$b \models_A \alpha_1 \vee \alpha_2$	iff $b \models_A \alpha_1$ or $b \models_A \alpha_2$
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 $i = j$ and $\sigma[i] \models_M \varphi$
$\mid \beta_1.\beta_2$	$\sigma[i,j] \models_M \beta_1.\beta_2$	iff $\exists k \in [i,j].\sigma[i,k] \models_M \beta_1$ and $\sigma[k,j] \models_M \beta_2$
$\mid \beta_1 \mid \beta_2$	$\sigma[i,j] \models_M \beta_1 \mid \beta_2$	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 $i = j$ or $\exists k > 0.\sigma[i,j] \models_M \beta_1^k$
State formulas:		
$\varphi ::=$ false	$s \models_M$ 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 $\sigma[i] \models_M \varphi_1$
$\mid \{\beta\}_{\geq p}$	$s \models_M \{\beta\}_{\geq p}$	iff $Pr_M(\{\sigma \in Paths_M(s) \mid \exists i \geq 0.\sigma[0,i] \models_M \beta\}) \geq p$
$\mid \{\beta\}_{> p}$	$s \models_M \{\beta\}_{> p}$	iff $Pr_M(\{\sigma \in Paths_M(s) \mid \exists i \geq 0.\sigma[0,i] \models_M \beta\}) > p$

Fig. 1: Modal and probabilistic operators over regular paths

A path $\sigma = s(= s_0) \stackrel{a_0}{\rightarrow} s_1 \stackrel{a_1}{\rightarrow} \cdots \stackrel{a_{n-1}}{\rightarrow} s_n \cdots$ 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 $0 \leq i \leq j$ is the path fragment $\sigma[i] \stackrel{a_i}{\rightarrow} \cdots \stackrel{a_{j-1}}{\rightarrow} \sigma[j]$, which reduces to the single-state path $\sigma[i]$ if $i = j$. The suffix of a path σ starting at state $\sigma[i]$ is noted σ_i. The set of paths going out of s is noted $Paths_M(s)$.

The probability space associated to a PTS M is defined similarly to DTMCs: the outcomes are paths and the basic events are the cylinder sets defined as $Cyl(s_0 \stackrel{a_0}{\rightarrow} \cdots \stackrel{a_{n-1}}{\rightarrow} s_n) = \{\sigma \in Paths_M(s_0) \mid \sigma[0,n] = s_0 \stackrel{a_0}{\rightarrow} \cdots \stackrel{a_{n-1}}{\rightarrow} s_n\}$, i.e., sets of paths sharing a common prefix. The σ-algebra associated to M is the smallest one containing all cylinder sets, and the probability measure of cylinder sets is defined as $Pr_M(Cyl(s_0 \stackrel{a_0}{\rightarrow} \cdots \stackrel{a_{n-1}}{\rightarrow} s_n)) = \Pi_{0 \leq i < n} \mathbf{P}(s_i, a_i, s_{i+1})$. By Caratheodory's extension theorem in classical measure theory [23], this probability measure uniquely extends to the σ-algebra, i.e., to all events (sets of paths) defined as countable unions of cylinder sets.

The regular probabilistic operator that we propose specifies 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*) [24], defined over the action formulas of ACTL (*Action-based CTL*) [49]. 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: true $= \neg$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: nil $=$ false* is the empty path 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 ($\{\,\}_{\geq p}$ and $\{\,\}_{>p}$) containing regular formulas. In line with the original definition of PCTL [29], we considered both strict ($> p$) and non strict ($\geq p$) conditions on the probability. Derived state operators can be defined as usual: true $= \neg$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, as propositional formulas denoting subsets of actions [49]. A path σ satisfies a regular formula β, noted $\sigma \models_M \beta$, if $\exists i \geq 0.\sigma[0,i] \models_M \beta$, i.e., it has a prefix belonging to the regular language defined by β. The testing operator $\varphi?$ of PDL specifies state formulas φ that must hold in the intermediate states of a path. Concatenation, choice, and transitive reflexive closure on regular formulas are defined in the standard way (β^k denotes the concatenation of β with itself k times). 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 $\{\beta\}_{\geq p}$ 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 $M = \langle S, A, T, \mathbf{P}, s^i\rangle$ 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).

We show below some examples of properties expressible in the logic. Safety properties, which express that "something bad never happens" [41], can be specified using the modality $[\beta]\,\mathsf{false}$, where the regular formula β denotes the undesirable paths and the box modality forbids their occurrence. Thus, mutual exclusion between two processes can be expressed as:

$$[\mathsf{true}^*.access_1.(\neg release_1)^*.access_2]\,\mathsf{false}$$

meaning that all paths containing an access of process 1 to a shared resource followed by an access of process 2 before process 1 has released the resource lead necessarily to states satisfying false, i.e., never occur.

The absence of deadlocks (sink states), a necessary condition for a well-defined PTS, can be specified by forbidding the paths leading to a deadlock (denoted by the modality $[\mathsf{true}]\,\mathsf{false}$), or alternatively by imposing that all reachable states have at least one successor (denoted by the modality $\langle\mathsf{true}\rangle\,\mathsf{true}$):

$$\neg\,\langle\mathsf{true}^*\rangle\,[\mathsf{true}]\,\mathsf{false} = [\mathsf{true}^*]\,\langle\mathsf{true}\rangle\,\mathsf{true}$$

The existential Until operator of CTL can be specified using a diamond modality as follows:

$$\mathsf{E}\,[\varphi_1\ \mathsf{U}\ \varphi_2] = \langle(\varphi_1?.\mathsf{true})^*\rangle\,\varphi_2$$

expressing that the current state has an outgoing path leading to a state satisfying φ_2 after zero or more transitions whose source states satisfy φ_1 (denoted by the regular subformula $\varphi_1?.\mathsf{true}$). Similarly, the existential Until operators of ACTL are specified as follows:

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

These action-based operators specify properties about both source states and actions of the intermediate transitions (denoted by the regular subformulas $\varphi_1?.\alpha_1$) and — for the second Until operator — also the last transition of the path fragment (denoted by the regular subformula $\varphi_1?.\alpha_2$).

The probabilistic operator expresses constraints on the probability of occurrence of certain paths described using regular formulas. The formula below specifies that the probability to send a message along an unreliable channel and receive it finally (possibly after a finite number of retries) is at least 90%:

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

By combining the modalities of PDL and the probabilistic operator, one can express quantitative response properties. The formula below specifies that every request of accessing a resource will be granted with probability 1 (i.e., almost surely):

$$[\mathsf{true}^*.request]\,\{\mathsf{true}^*.grant\}_{\geq 1}$$

The operator $\{\beta\}_{\geq p}$ generalizes naturally the Until operators of classical probabilistic branching-time logics. The Until operator of PCTL [29] without discrete time (i.e., without the step-bounding clause $\mathsf{U}^{\leq t}$), and probabilistic versions of the two Until operators of ACTL are expressed as follows:

$$[\varphi_1\ \mathsf{U}\ \varphi_2]_{\geq p} = \{(\varphi_1?.\mathsf{true})^*.\varphi_2?\}_{\geq p}$$

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

These encodings consider the Until operators as a means of describing path fragments, and reformulate them using regular operators, which are strictly more expressive: for instance, the regular formula $(\mathsf{true}.\mathsf{true}.\varphi?)^+$ denoting the path fragments in which φ holds at even states is not expressible using Until operators [50]. Our extension of regular formulas with data handling (see Section 4) will enable to express also the step-bounded Until operators, and therefore to subsume full P(A)CTL.

Measurability of β-events. The probabilistic operator $\{\beta\}_{\geq p}$ interpreted on a state s refers to the probability measure of the set of paths going out of s and having a prefix satisfying β, defined as $Paths(s,\beta) = \{\sigma \in Paths(s) \mid \sigma \models_M \beta\}$. For this semantics to be well-defined, the set $Paths(s,\beta)$ must be measurable, i.e., it must be an event of the σ-algebra associated to the PTS M. This follows directly from the definition of $\{\beta\}_{\geq p}$ in Figure 1, because $Paths(s,\beta)$ can be considered as a countable union of cylinder sets:

$$\begin{aligned}Paths(s,\beta) &= \{\sigma \in Paths(s) \mid \sigma \models_M \beta\}\\ &= \{\sigma \in Paths(s) \mid \exists i \geq 0.\sigma[0,i] \models_M \beta\}\\ &= \bigcup_{\substack{i \geq 0\\ \sigma[0,i] \in Paths_{fin}(s)\\ \sigma[0,i] \models_M \beta}} Cyl(\sigma[0,i])\end{aligned}$$

where $Paths_{fin}(s)$ is the set of finite paths going out of s. The countable union above can be computed by enumerating, for each $i \geq 0$, all finite path fragments $\sigma[0,i] \in Paths_{fin}(s)$ of length i and checking whether they satisfy β. This last check can be done by verifying the PDL formula $\langle\beta\rangle\,[\mathsf{true}]\,\mathsf{false}$ on s in the LTS consisting solely of the finite path fragment $\sigma[0,i]$ by using, e.g., the model checking procedure given in [22].

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 two steps can be found in [47], and of the third and fourth steps (in a state-based setting) in [44].

We illustrate the model checking method using the classical example of a six-sided dice emulated using a fair coin [33]. First, we consider a dataless version of the system (i.e., in which actions are simply names), represented as a PTS in Figure 2(a). The actions of tossing a head or tail of the coin are denoted by *head* and *tail*, each one having 0.5 probability. The actions corresponding to the dice faces are denoted by $dice_i$ for $1 \leq i \leq 6$, each one occurring with probability 1. We will check on this PTS the formula Ψ_1, stating that the probability of reaching the dice side marked with four directly after tossing the last head (if any) is at least 1/6:

$$\Psi_1 = \{(\mathsf{true}^*.head)^*.dice_4\}_{\geq 0.16}$$

Strictly speaking, this formula could be verified using a PACTL model checker, since the regular subformula can be expressed equivalently without nested star operators as $dice_4|(\mathsf{true}^*.head.dice_4)$. The existence of a path matching it can be specified in ACTL using two Until operators: $\mathsf{E}\,[\mathsf{true}_{\mathsf{false}}\ \mathsf{U}_{dice_4}\ \mathsf{true}] \vee \mathsf{E}\,[\mathsf{true}_{\mathsf{true}}\ \mathsf{U}_{head}\ \mathsf{E}\,[\mathsf{true}_{\mathsf{false}}\ \mathsf{U}_{dice_4}\ \mathsf{true}]]$. This would require the model checker to compute the probability measures of each Until operator, then check that their sum is at least 0.16. However, we use formula Ψ_1 for the sake of simplicity to illustrate our model checking procedure. Examples of more complex properties, most of them unexpressible in PACTL, can be found in Section 6.

1. Translation to PDL with recursion. To evaluate an operator $\{\beta\}_{\geq p}$ on a PTS $M = \langle S, A, T, \mathbf{P}, s^i \rangle$ 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 was introduced in [47] for model checking PDL formulas. 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 $\rho : \mathcal{X} \to 2^S$, 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:

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

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

In the sequel, we consider PDLR specifications in *der**ivative** **n**ormal **f**orm* (RNF), which are the modal logic counterparts of Brzozowski's (generalized) derivatives of regular expressions [10]:

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

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:

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

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:

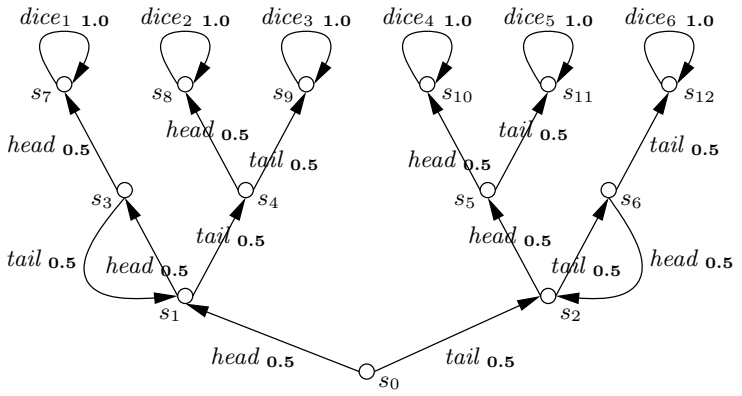
$$\{P_1(\sigma) = \exists l \geq 0.(\sigma[0, l] \models \beta \wedge P_2(\sigma_l)) \qquad P_2(\sigma) = \mathsf{true}\}$$

According to the interpretation of regular formulas (see Fig. 1), the path predicate $P_1(\sigma)$ holds iff $\sigma \models \beta$, and also $(\mu\Phi)_1 = \{s \in S \mid \exists \sigma \in Paths_M(s).\sigma \models \beta\}$.

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 [38], which contains only HML modalities on action formulas. Regular operators can be eliminated by applying the following substitutions, which are valid equalities in PDL [24]:

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

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



(a) PTS (dataless version)

$$\left\{ \begin{array}{l} X_1 = \langle (\mathsf{true}^*.head)^*.dice_4 \rangle\, X_2 \\ X_2 = \mathsf{true} \end{array} \right\}$$

(b) PDLR

$$\left\{ \begin{array}{l} X_1 = \langle \mathsf{nil} \rangle\, X_4 \\ X_2 = \mathsf{true} \\ X_3 = \langle dice_4 \rangle\, X_2 \\ X_4 = \langle \mathsf{nil} \rangle\, X_3 \vee \langle \mathsf{nil} \rangle\, X_5 \\ X_5 = \langle \mathsf{nil} \rangle\, X_7 \\ X_6 = \langle head \rangle\, X_4 \\ X_7 = \langle \mathsf{nil} \rangle\, X_6 \vee \langle \mathsf{true} \rangle\, X_7 \end{array} \right\}$$

(c) HMLR

$$\left\{ \begin{array}{l} X_1 = \langle dice_4 \rangle\, X_2 \vee \langle head \rangle\, X_1 \vee \langle \mathsf{true} \rangle\, X_7 \\ X_2 = \mathsf{true} \\ X_7 = \langle head \rangle\, X_1 \vee \langle \mathsf{true} \rangle\, X_7 \end{array} \right\}$$

(d) guarded HMLR

$$\left\{ \begin{array}{l} X_1 = \langle dice_4 \rangle\, X_2 \vee \langle head \rangle\, X_1 \vee \\ \qquad \langle \neg (head \vee dice_4) \rangle\, X_7 \\ X_2 = \mathsf{true} \\ X_7 = \langle head \rangle\, X_1 \vee \langle \neg head \rangle\, X_7 \end{array} \right\}$$

(e) deterministic HMLR

$$\left\{ \begin{array}{l} Z_{1,0} = 0.5 \cdot \boxed{Z_{1,1}} + 0.5 \cdot Z_{7,2} \\ Z_{7,2} = 0.5 \cdot Z_{1,5} + 0.5 \cdot Z_{7,6} \\ Z_{1,5} = 0.5 \cdot Z_{1,10} + 0.5 \cdot \boxed{Z_{7,11}} \\ Z_{1,10} = 1.0 \\ Z_{7,6} = 0.5 \cdot Z_{1,2} + 0.5 \cdot \boxed{Z_{7,12}} \\ Z_{1,2} = 0.5 \cdot Z_{1,5} + 0.5 \cdot Z_{7,6} \end{array} \right\}$$

(f) LES

$$\left\{ \begin{array}{l} \boxed{X_{1,1}} = X_{1,3} \vee X_{7,4} \\ X_{1,3} = X_{1,7} \\ X_{1,7} = X_{7,7} \\ X_{7,4} = X_{1,8} \vee X_{7,9} \\ X_{1,8} = X_{7,8} \\ X_{7,7} = X_{7,7} \\ X_{7,8} = X_{7,8} \\ X_{7,9} = X_{7,9} \\ \boxed{X_{7,11}} = X_{7,11} \\ \boxed{X_{7,12}} = X_{7,12} \end{array} \right\}$$

(g) BES (excerpt)

Fig. 2: Model checking formula $\Psi_1 = \{(\mathsf{true}^*.head)^*.dice_4\}_{\geq 0.16}$ on the PTS simulating the six-sided dice using a coin

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 $\langle \alpha_{ij} \rangle\, X_{ij}$ and $\langle \mathsf{nil} \rangle\, X_{ij}$ (equivalent to X_{ij}), which correspond to *guarded* and *unguarded* occurrences of variables X_{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 [34] can be specialized for HMLR specifications by applying the following actions for each equation defining X_i:

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

tion i, and rearrange the right hand sides to maintain the equations in RNF.

This produces a guarded HMLR specification:

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

which is the exact modal logic counterpart of Brzozowski's derivatives of regular expressions [10] defined on the alphabet of action formulas. The transformation to guarded form generally decreases the number of equations in the HMLR specification (by producing unreachable and/or duplicate equations, which are eliminated), but may increase the number of operators in the right hand sides.

4. Determinization. A guarded HMLR specification may contain, in the right hand side of an equation i, several *alternatives* $\varphi_{ij} \wedge \langle \alpha_{ij} \rangle X_{ij}$ whose guards φ_{ij} and/or action formulas α_{ij} are not disjoint, i.e., they can match the same transition going out of the current state. This is 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 equations. This can be done by applying the classical subset construction, which produces a deterministic HMLR specification defined on meta-variables (i.e., sets of propositional variables):

$$\left\{ \begin{array}{l} X_I = \bigvee_{\emptyset \subset J \subseteq alt(I)} \bigvee_{\emptyset \subset L \subseteq J} (\\ \qquad (\bigwedge_{ij \in J} \varphi_{ij} \wedge \bigwedge_{i'j' \in alt(I) \setminus J} \neg\, \varphi_{i'j'}) \\ \qquad \wedge \\ \qquad \langle \bigwedge_{kl \in L} \alpha_{kl} \wedge \bigwedge_{k'l' \in J \setminus L} \neg\, \alpha_{k'l'} \rangle X_L \\ \quad) \\ \quad \vee \\ \quad \bigvee_{i \in I} \varphi_i \end{array} \right\}_{I \subseteq [1,n]}$$

where $I \subseteq [1,n]$ is a subset of the variable indices and X_I is a shorthand notation for the meta-variable $\{X_i \mid i \in I\}$. Given a subset I, the set of indices of alternatives contained in the equations defining variables in X_I is $alt(I) = \{ij \mid i \in I \wedge j \in [1, n_i]\}$. A transition $s \xrightarrow{a} s'$ matches an alternative $\varphi_{ij} \wedge \langle \alpha_{ij} \rangle X_{ij}$ if $s \models \varphi_{ij}$, $a \models \alpha_{ij}$, and $s' \models X_{ij}$. Intuitively, the right hand side of the meta-equation defining X_I is obtained by considering all situations when a transition $s \xrightarrow{a} s'$ matches a subset of the alternatives $alt(I)$ and requiring that the target state s' satisfies the meta-variable induced by the matching alternatives. For a transition $s \xrightarrow{a} s'$ to match a subset of alternatives $J \subseteq alt(I)$, state s must satisfy all guards φ_{ij} of alternatives in J, and none of the guards of alternatives in $alt(I) \setminus J$. Then, for all modalities $\langle \alpha_{ij} \rangle X_{ij}$ contained in the alternatives in J, it may be the case that a satisfies only the action formulas contained in a subset $L \subseteq J$ of alternatives (and none of the action formulas of the alternatives in $J \setminus L$), and therefore s' must satisfy the meta-variable X_L. As shown in [44] 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 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 that the occurrences of the *head* action are separated by actions other than *head* and by taking into account that, in the PTS of Figure 2(a), a *dice* action cannot occur before a *head*: $((\neg head)^*.head)^*.dice_4$. In practice, the size of the determinized HMLR specification can be reduced by eliminating duplicate equations and by identifying 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 (in which the meta-variables have been renamed into ordinary propositional variables) corresponding to a regular formula β:

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

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

$$\left\{ \begin{array}{l} P_i(\sigma) = \bigvee_{j=1}^{n_i} (\sigma[0] \models \varphi_{ij} \wedge \sigma_a[0] \models \alpha_{ij} \wedge \\ \qquad\qquad P_{ij}(\sigma_1)) \\ \qquad \vee \\ \qquad \sigma[0] \models \varphi_i \end{array} \right\}_{1 \leq i \leq n}$$

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:

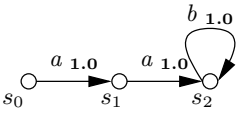
$$\left\{ \begin{array}{l} Z_{i,s} = \mathsf{if}\ \boxed{s \not\models X_i}\ \mathsf{then}\ 0 \\ \qquad \mathsf{else\ if}\ s \models \varphi_i\ \mathsf{then}\ 1 \\ \qquad \mathsf{else} \\ \qquad\quad \sum_{j=1}^{n_i} \mathsf{if}\ s \not\models \varphi_{ij}\ \mathsf{then}\ 0 \\ \qquad\qquad \mathsf{else} \\ \qquad\qquad\quad \sum_{s \xrightarrow{a} s'} \mathbf{P}(s,a,s') \cdot Z_{ij,s'} \\ \qquad\qquad a \models \alpha_{ij} \end{array} \right\} \begin{array}{l} \\ \\ \\ \\ \\ \\ \\ 1 \le i \le n \\ s \in S \end{array} \quad (*)$$

$$\left\{ \begin{array}{l} X_{i,s} = \bigvee_{j=1}^{n_i} \big(s \models \varphi_{ij} \wedge \\ \qquad\qquad \bigvee_{s \xrightarrow{a} s'} \big(a \models \alpha_{ij} \wedge X_{ij,s'} \big) \\ \qquad \big) \\ \qquad \vee \\ \qquad s \models \varphi_i \end{array} \right\} \begin{array}{l} \\ \\ \\ \\ \\ 1 \le i \le n \\ s \in S \end{array} \quad (**)$$

The LES $(*)$ is obtained by a translation similar to the classical one defined originally for PCTL [29]. A numerical variable $Z_{i,s}$ denotes the probability measure of the paths going out of state s and satisfying the path predicate P_i. The BES $(**)$, with minimal fixed point semantics, is produced by the classical translation employed for model checking modal μ-calculus formulas on LTSs [16,3]. 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 Z_{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 for evaluating the closed state formulas φ_{ij}, φ_i on state s.

The determinization of the HMLR specification guarantees that the sum of coefficients in the right hand side of each equation of the LES is at most 1. If the HMLR specification is not determinized, such as the one shown in Figure 3(c), the LES encoding the interpretation of the formula on the PTS in Figure 3(a) has the wrong solution $Z_{1,0} = 2.0$ (variable $Z_{1,2} = 0$ since $s_2 \not\models X_1$, and variables $Z_{2,1} = Z_{2,2} = 1$ since $X_2 = \mathsf{true}$). This is because the nondeterministic equation defining X_1 in the HMLR specification matches twice the suffix of the path going out of s_0. Note that determinization ensures a sound LES translation also if some states have several outgoing transitions labeled by the same action a.

The condition $s \not\models X_i$ surrounded in the LES (*) has the role of pruning the exploration of the PTS if the current state s does not have any outgoing path matching the suffix of β denoted by X_i. As explained in [7, Chap. 10], this is necessary to avoid singularities in the



$$\left\{ \begin{array}{l} X_1 = \langle a \rangle\, X_2 \vee \langle \mathsf{true} \rangle\, X_1 \\ X_2 = \mathsf{true} \end{array} \right\}$$

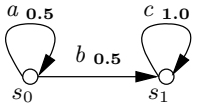
(a) PTS (c) guarded HMLR

$\{\mathsf{true}^*.a\}_{\geq 0}$

$$\left\{ \begin{array}{l} Z_{1,0} = 1.0 \cdot Z_{2,1} + 1.0 \cdot Z_{1,1} \\ Z_{1,1} = 1.0 \cdot Z_{2,2} + 1.0 \cdot \boxed{Z_{1,2}} \end{array} \right\}$$

(b) formula (d) LES

Fig. 3: Nondeterministic regular formula



$$\left\{ \begin{array}{l} X_1 = \langle a \rangle\, X_2 \vee \langle \neg a \rangle\, X_1 \\ X_2 = \mathsf{true} \end{array} \right\}$$

(a) PTS (c) deterministic HMLR

$\{(\neg a)^*.a\}_{\geq 0}$

$$\left\{ \begin{array}{l} Z_{1,0} = 0.5 \cdot Z_{2,0} + 0.5 \cdot Z_{1,1} \\ Z_{1,1} = 1.0 \cdot Z_{1,1} \end{array} \right\}$$

(b) formula (d) LES

Fig. 4: Unreachable path suffix

LES and ensure a unique solution. For example, if we neglect the condition $s \not\models X_i$ when checking the deterministic HMLR specification shown in Figure 4(c) on the PTS shown in Figure 4(a), the resulting LES contains the singular equation $Z_{1,1} = Z_{1,1}$. This is precisely because $s_1 \not\models X_1$, meaning that from s_1 there is no path matching the regular formula suffix denoted by X_1.

By solving the LES obtained in Figure 2(f), we obtain $Z_{1,0} = 1/6$, meaning that tossing the head of the coin an (arbitrarily large) number of times corresponds to rolling the four-side of the dice with 1/6 probability, and therefore the formula Ψ_1 is valid on the PTS. The numerical variables $Z_{1,1}$, $Z_{7,11}$, and $Z_{7,12}$ surrounded by boxes in Figure 2(f) are equal to 0, since the path suffixes denoted by the propositional variables X_1 and X_7 are not reachable from states s_1, s_{11}, and s_{12} of the PTS; this is checked by solving the Boolean variables $X_{1,1}$, $X_{7,11}$, and $X_{7,12}$ of the BES in Figure 2(g), which are equal to false. If we evaluate the Ψ_1 formula for every $dice_i$ action, we obtain a probability equal to 1/6 for $i \in \{1,2,4\}$ and to 0 for $i \in \{3,5,6\}$, the latter sides of the dice being not directly reachable after a *head* action.

4 Extension with data handling

The regular formulas that we used so far belong to the dataless fragment [47] 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 [48] 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 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.

Expressions:

$$e ::= x \mid f(e)$$

$$\begin{aligned} [\![x]\!]\,\delta &= \delta(x) \\ [\![f(e)]\!]\,\delta &= f([\![e]\!]\,\delta) \end{aligned}$$

Action patterns:

$$\alpha ::= \{c\ !e\} \mid \{c\ ?x{:}T\}$$

$$\begin{aligned} [\![\{c\ !e\}]\!]\,\delta &= \{c\ [\![e]\!]\,\delta\} \\ [\![\{c\ ?x{:}T\}]\!]\,\delta &= \{c\ v \mid v \in T\} \\ env_{c\ v}(\{c\ ?x{:}T\}) &= [v/x] \text{ if } v \in T \\ env_a(\alpha) &= [\,] \text{ otherwise} \end{aligned}$$

State formulas:

$$\varphi ::= \langle\alpha\rangle\,\varphi \mid Y(e) \mid \mu Y(x{:}T{:=}e).\varphi$$

$$\begin{aligned} [\![\langle\alpha\rangle\,\varphi]\!]\,\rho\delta &= \{s \in S \mid \exists s \xrightarrow{a} s'.a \in [\![\alpha]\!]\,\delta \wedge \\ &\qquad s' \in [\![\varphi]\!]\,\rho(\delta \oslash env_a(\alpha))\} \\ [\![Y(e)]\!]\,\rho\delta &= \rho(Y)([\![e]\!]\,\delta) \\ [\![\mu Y(x{:}T{:=}e).\varphi]\!]\,\rho\delta &= \mu\Phi_{\rho\delta}([\![e]\!]\,\delta) \end{aligned}$$

where $\Phi_{\rho\delta} : (T \to 2^S) \to (T \to 2^S)$,
$(\Phi_{\rho\delta}(F))(v) = [\![\varphi]\!]\,(\rho \oslash [F/Y])(\delta \oslash [v/x])$

Fig. 5: Syntax and semantics of basic MCL operators

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 values contained in actions, MCL provides several basic constructs, defined in Figure 5. Data expressions e are built over data variables x and functions f (restricted, for simplicity, to a single argument in Fig. 5). The interpretation of e is defined w.r.t. a data context δ assigning values to all data variables occurring in e.

Actions containing data are specified using *action predicates* of the form "$\{c\ !e\}$" (value matching) or "$\{c\ ?x{:}T\}$" (value extraction). The former predicate matches an action $c\ v$ with v equal to the value of e, whereas the latter predicate also captures the value v (which must be of type T) and stores it in the variable x. The successful evaluation of a predicate "$\{c\ ?x{:}T\}$" on an action a also produces a data context $env_a(\{c\ ?x{:}T\})$ where x is assigned the value extracted from a. The action predicates of MCL are slightly more involved than those shown in Figure 5: several clauses "$!e$" and "$?x{:}T$" can be freely combined (the variables defined in the extraction clauses are visible inside the action predicate and also in the enclosing formula), the "..." wildcard can be used to match zero or more values, and an optional clause "where b" can be used to specify a Boolean condition b over the extracted variables. An action satisfies an action predicate if its structure is compatible with the clauses of the predicate and the "where" guard (if present) evaluates to true in the context of the data variables extracted from that action.

The basic state formulas of MCL are built from Boolean connectors, modalities, and parameterized fixed point operators (restricted, for simplicity, to a single parameter in Fig. 5). A fixed point formula $\mu Y(x{:}T{:=}e).\varphi$ denotes both the definition and the call *in situ* of a propositional variable Y parameterized by x. Its interpretation w.r.t. a propositional context ρ and a data context δ is the call of $\mu\Phi_{\rho\delta}$ with the value of e in the context δ, $\mu\Phi_{\rho\delta}$ being the minimal solution of the equation $Y(x) = \varphi$ interpreted w.r.t. the contexts ρ and δ.

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 Subsection 4.2), and definition of variables ("let"). Finally, the state formulas of MCL also provide modalities containing regular formulas, quantifiers over finite domains, and programming language constructs ("if" and "let") [48].

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:

$$\begin{aligned} \beta ::= &\ \mathsf{loop}\ (x{:}T{:=}e_0) : (x'{:}T')\ \mathsf{in} \\ &\quad \beta \\ &\ \mathsf{end\ loop} \\ \mid &\ \mathsf{continue}\ (e) \\ \mid &\ \mathsf{exit}\ (e') \end{aligned}$$

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. The iteration variable x, which is visible inside the loop body 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 conciseness, we define the semantics of the "loop" operator by translating it to basic MCL in the context of an enclosing diamond modality (a dual translation holds for an enclosing box modality). The translation $loop2mcl_{Z/x:T/x':T'}(\varphi)$ is defined in Figure 6, where x and x' are the iteration and return data variables of the immediately enclosing "loop", and Z is the propositional variable associated to it. 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 calls of the propositional variable with the corresponding arguments, and the occurrences of "exit" are translated into "let" state formulas defining the return variables and assigning them the corresponding return values. The rules for the other regular operators generalize those of the translation from PDL to modal μ-calculus [22].

All iteration operators on MCL regular formulas can be expressed in terms of the "loop" operator, as shown in Table 1. For simplicity, we omitted the definition of β^*, β^+, $\beta\{e\}$ (iteration e times), and $\beta\{... e\}$ (iteration at most e times), which are equivalent to $\beta\{0 ...\}$, $\beta\{1 ...\}$, $\beta\{e ... e\}$, and $\beta\{0 ... e\}$, 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 encoding bounded iteration as a "loop" and applying the translation rules of general iteration, we obtain:

Table 1: Encoding MCL iteration operators using "loop"

Syntax	Meaning	Encoding using "loop"

$\beta\{e ...\}$	$\geq e$ times	loop (c:nat := e) in
		  if $c > 0$ then
		  β . continue $(c-1)$
		  else
		  exit | β . continue (c)
		  end if
		end loop

$\beta\{e_1 ... e_2\}$	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

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

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

The bounded iteration operators $\beta\{e\}$, $\beta\{e ...\}$, and $\beta\{e_1 ... e_2\}$ 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 (where $t \geq 0$ is the number of ticks until φ_2):

$$\left[\varphi_1\ \mathsf{U}^{\leq t}\ \varphi_2\right]_{\geq p} = \{(\varphi_1?.\mathsf{true})\{0 ... t\}.\varphi_2?\}_{\geq p}$$

$$\left[\varphi_{1\alpha_1}\ \mathsf{U}^{\leq t}\ \varphi_2\right]_{\geq p} = \{(\varphi_1?.\alpha_1)\{0 ... t\}.\varphi_2?\}_{\geq p}$$

$$\left[\varphi_{1\alpha_1}\ \mathsf{U}^{\leq t}_{\alpha_2}\ \varphi_2\right]_{\geq p} = \{(\varphi_1?.\alpha_1)\{0 ... t\}.\varphi_1?.\alpha_2.\varphi_2?\}_{\geq p}$$

$$loop2mcl_{Z/x{:}T/x'{:}T'}\left(\left\langle \begin{array}{l}\mathsf{loop}\ (x{:}T{:=}e_0) : (x'{:}T')\ \mathsf{in}\\ \quad \beta\\ \mathsf{end\ loop}\end{array}\right\rangle \varphi\right) \stackrel{\text{def}}{=} \mu Y(x{:}T{:=}e_0).loop2mcl_{Y/x:T/x'{:}T'}(\langle\beta\rangle\,\varphi)$$

$$loop2mcl_{Z/x:T/x'{:}T'}\,(\langle\mathsf{continue}\ (e)\rangle\,\varphi) \stackrel{\text{def}}{=} Z(e)$$

$$loop2mcl_{Z/x:T/x'{:}T'}\,(\langle\mathsf{exit}\ (e')\rangle\,\varphi) \stackrel{\text{def}}{=} \mathsf{let}\ x'{:}T' := e'\ \mathsf{in}\ loop2mcl_{Z/x:T/x'{:}T'}(\varphi)\ \mathsf{end\ let}$$

$$loop2mcl_{Z/x:T/x'{:}T'}(\langle\alpha\rangle\,\varphi) \stackrel{\text{def}}{=} \langle\alpha\rangle\,loop2mcl_{Z/x:T/x'{:}T'}(\varphi)$$

$$loop2mcl_{Z/x:T/x'{:}T'}(\langle\varphi_1?\rangle\,\varphi) \stackrel{\text{def}}{=} \langle loop2mcl_{Z/x:T/x'{:}T'}(\varphi_1)?\rangle\,loop2mcl_{Z/x:T/x'{:}T'}(\varphi)$$

$$loop2mcl_{Z/x:T/x'{:}T'}(\langle\beta_1|\beta_2\rangle\,\varphi) \stackrel{\text{def}}{=} loop2mcl_{Z/x:T/x'{:}T'}(\langle\beta_1\rangle\,\varphi) \vee loop2mcl_{Z/x:T/x'{:}T'}(\langle\beta_2\rangle\,\varphi)$$

$$loop2mcl_{Z/x:T/x'{:}T'}(\langle\beta_1.\beta_2\rangle\,\varphi) \stackrel{\text{def}}{=} loop2mcl_{Z/x:T/x'{:}T'}(\langle\beta_1\rangle\,loop2mcl_{Z/x:T/x'{:}T'}(\langle\beta_2\rangle\,\varphi))$$

$$loop2mcl_{Z/x:T/x'{:}T'}(\langle\beta^*\rangle\,\varphi) \stackrel{\text{def}}{=} \mu Y.(loop2mcl_{Z/x:T/x'{:}T'}(\varphi) \vee loop2mcl_{Z/x:T/x'{:}T'}(\langle\beta\rangle\,Y))$$

Fig. 6: Translation of the general iteration operators in basic MCL

Besides counting, the "loop" operator can characterize complex paths in a PTS, by collecting the data values (costs) present on actions and using them in arbitrary computations, as it will be illustrated in Section 6.

4.3 Model checking method with data handling

The on-the-fly model checking method described 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 on a data handling version of the PTS modeling the six-sided dice using a fair coin, shown on Figure 7(a). The actions of tossing the head and tail of the coin are denoted by "*toss* 1" and "*toss* 0", and the actions corresponding to the dice faces are denoted by "*dice* i" for $1 \leq i \leq 6$. We will check on this PTS the formula Ψ_2, stating that the probability of reaching the dice side marked with one by tossing first a side of the coin and then tossing the same side at most n times (where n is a parameter to be instantiated) is at least 10%:

$$\begin{array}{ll}\Psi_2 = \{ & \{toss\ ?v{:}\mathsf{nat}\}.\\ & ((\neg\{toss\ !v\})^*.\{toss\ !v\})\{...\ n\}.\\ & \{dice\ !1\}\quad \}_{\geq 0.1}\end{array}$$

The various translation phases are illustrated on Figure 7. The translation rules for standard regular operators given in Section 3, extended with data handling, are applied for eliminating the "." operators in the PDLR specification. To facilitate the subsequent model checking steps, the new equations introduced by these rules are data-closed, i.e., the propositional variables in their left hand sides are parameterized by the data variables occurring in their right hand side formulas. For example, when translating the right hand side of the equation defining X_1, the extended rule applied is $\langle\{toss\ ?v{:}\mathsf{nat}\}.\beta\rangle\,X_2 = \langle\{toss\ ?v{:}\mathsf{nat}\}\rangle\,X_3(v)$, where $X_3(v) = \langle\beta\rangle\,X_2$, ensuring that data variable v occurring in β is a parameter of X_3. Then, the iteration at most n times is translated into a "loop" operator, 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 c and by the value v of the coin side captured by the first action of the regular formula (Fig. 7(c)).

This HMLR specification is then brought to guarded form (Fig. 7(d)) by carrying out systematically the appropriate substitutions and keeping track of the data parameters. For example, variable $X_3(v)$ is replaced by $X_5(v,n)$ in the equation of X_1, and $X_6(v,c)$ is replaced by $\langle\{toss\ !v\}\rangle\,X_5(v,c-1)$ in the equation defining $X_7(v,c)$.

The guarded HMLR specification is determinized by ensuring that the alternatives in the right hand side of equation defining $X_5(v,c)$ are mutually exclusive (all the other equations being already deterministic). This produces the alternatives $\langle\{dice\ !1\} \wedge \neg\{toss\ !v\}\rangle\,\{X_2, X_7(v,c)\}$ and $\langle\neg\{toss\ !v\} \wedge \neg\{dice\ !1\}\rangle\,\{X_7(v,c)\}$. The former alternative is equivalent to $\langle\{dice\ !1\}\rangle\,\{X_2\}$ because $\{dice\ !1\} \Rightarrow \neg\{toss\ !v\}$ and X_2 denotes the final states of the paths satisfying the initial regular formula. The other alternatives present in the equations led to singleton meta-variables, which are replaced by the variables themselves (Fig. 7(e)).

Next, a parameterized LES is produced (Fig. 7(f)) by the translation scheme given in Section 3, extended to handle data parameters. For instance, variable $Z_{5,1}(1,n)$

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 parameters v and c set to 1 and n, respectively. The numerical variables $Z_{5,2}(0,n)$ and $Z_{7,4}(1,c)$ surrounded by boxes in Figure 7(f) are equal to 0, since the path suffixes denoted by the propositional variables $X_5(0,n)$ and $X_7(1,c)$ are not reachable from states s_2 and s_4 of the PTS (this is checked by solving the corresponding Boolean variables of the BES, not shown in Figure 7).

Finally, a plain LES is generated (Fig. 7(g)) by instantiating $n = 2$ in the parameterized LES. By solving this LES (e.g., using substitution), we obtain $Z_{1,0} = 0.125$, 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, tossing first a side of the coin, then tossing again the same side at most $n = 2$ times corresponds with 12.5% probability to rolling the one-side of the dice, and therefore the Ψ_2 formula is valid on the PTS in Figure 7(a). If we evaluate the Ψ_2 formula with $n = 2$ for every action "*dice i*", we obtain a probability equal to 0.125 for $i \in \{1, 6\}$, to 0.156 for $i \in \{2, 5\}$, and to 0 for $i \in \{3, 4\}$, the latter sides of the dice being not reachable after tossing once, then at most $n = 2$ times the same side of the coin.

4.4 Soundness and termination of the method

The presence of data parameters (with infinite domains) in the regular formulas β makes necessary two additional hypotheses about the structure of β in order to ensure the feasibility of our model checking method for the probabilistic operator $\{\beta\}_{\geq p}$.

Soundness of the translation to guarded form. The translation of (parameterized) HMLR specifications to guarded form relies on the absorption property for μ-calculus formulas with data parameters:

$$\mu X(u).(X(u) \vee \varphi) = \mu X(u).\varphi$$

where φ may contain occurrences of the propositional variable X with arbitrary data arguments. The above equality does not hold anymore if the unguarded occurrence of X in the left hand side has an argument e different from the parameter u. Typically, such cases correspond to regular formulas that perform pure computations on data without exploring any transition in the PTS. For example, the following regular modality specifies the classical primality testing of a natural number n (odd and greater than 3) by checking that n is not divided by any of the odd numbers from 3 to $\sqrt{n}$:

$$\begin{array}{l}\langle\ \mathsf{loop}\ (k{:}\mathsf{nat}{:=}3) : (r{:}\mathsf{bool})\ \mathsf{in}\\ \quad \mathsf{if}\ k*k > n\ \mathsf{then}\\ \quad\quad \mathsf{exit}\ (\mathsf{true})\\ \quad \mathsf{elsif}\ n\%k = 0\ \mathsf{then}\\ \quad\quad \mathsf{exit}\ (\mathsf{false})\\ \quad \mathsf{else}\\ \quad\quad \mathsf{continue}\ (k+2)\\ \quad \mathsf{end\ if}\\ \mathsf{end\ loop}\ \rangle\ r\end{array}$$

The corresponding HMLR specification, shown below, contains the unguarded occurrence $X_2(k+2)$, which cannot be eliminated by absorption:

$$\left\{\begin{array}{l} X_1 = X_2(3)\\ X_2(k) = \mathsf{if}\ k*k > n\ \mathsf{then}\\ \qquad\qquad \mathsf{true}\\ \qquad\quad \mathsf{elsif}\ n\%k = 0\ \mathsf{then}\\ \qquad\qquad \mathsf{false}\\ \qquad\quad \mathsf{else}\\ \qquad\qquad X_2(k+2)\\ \qquad\quad \mathsf{end\ if}\end{array}\right\}$$

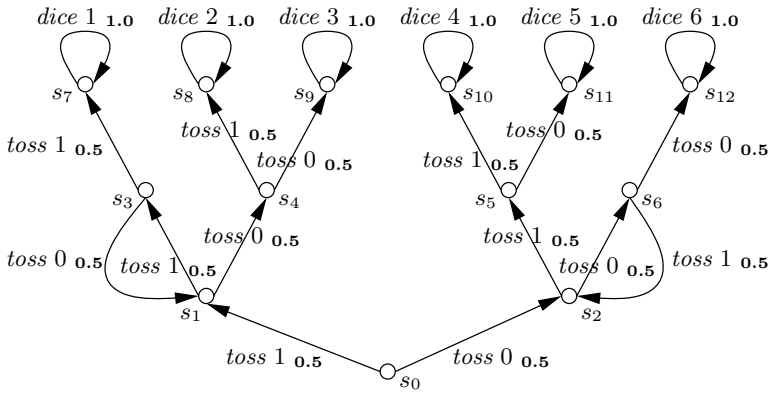
In practice, we forbid such kind of regular formulas purely operating on data, since they can be replaced by functions, and we reserve regular formulas for the specification of (non empty) paths in the PTS.

Termination of the LES and BES construction. After translating the regular formula into a (parameterized) deterministic HMLR specification, the last phase of the model checking procedure consists in generating and solving on the fly the LES and BES. These systems depend not only on the (finite) information contained in the PTS, but also on the values of the data parameters (with infinite domains) created during instantiation. Therefore, the convergence of the whole procedure relies on the termination of the instantiation phase, which must create finite LES and BES. This is in general undecidable, similarly to the termination of term rewriting [19]. Typically, such situations happen for "pathological" formulas, which carry on divergent computations unrelated to the data values contained in the PTS actions. For example, the following modality:

$$\begin{array}{l}\langle\ \mathsf{loop}\ (k{:}\mathsf{nat}{:=}0)\ \mathsf{in}\\ \quad a\ .\ \mathsf{continue}\ (k+1)\\ \mathsf{end\ loop}\ \rangle\ \mathsf{true}\end{array}$$

will not converge on the PTS consisting of a single loop $s \xrightarrow{a\ \mathbf{1.0}} s$, since it will entail the construction of an infinite LES $\{Z_s(0) = Z_s(1), Z_s(1) = Z_s(2), ...\}$. In practice, for such divergent formulas, the instantiation phase will attempt to generate an infinite LES, eventually causing the whole model checking procedure to abort when the available memory is exhausted.

The regular formulas β used in the probabilistic operator are meant to characterize *finite* paths in the PTS.



(a) PTS (data handling version)

$$\left\{\begin{array}{l} X_1 = \langle\{toss\ ?v{:}\mathsf{nat}\}. \\ \qquad ((\neg\{toss\ !v\})^*. \\ \qquad\ \{toss\ !v\})\{...\ n\}. \\ \qquad \{dice\ !1\}\rangle X_2 \\ X_2 = \mathsf{true} \end{array}\right\}$$

(b) PDLR

$$\left\{\begin{array}{rl} X_1 = & \langle\{toss\ ?v{:}\mathsf{nat}\}\rangle\, X_3(v) \\ X_2 = & \mathsf{true} \\ X_3(v) = & \langle\mathsf{nil}\rangle\, X_5(v,n) \\ X_4 = & \langle\{dice\ !1\}\rangle\, X_2 \\ X_5(v,c) = & \mathsf{if}\ c>0\ \mathsf{then} \\ & \quad \langle\mathsf{nil}\rangle\, X_4 \vee \\ & \quad \langle\mathsf{nil}\rangle\, X_7(v,c) \\ & \mathsf{else} \\ & \quad \langle\mathsf{nil}\rangle\, X_4 \\ & \mathsf{end\ if} \\ X_6(v,c) = & \langle\{toss\ !v\}\rangle\, X_5(v,c-1) \\ X_7(v,c) = & \langle\mathsf{nil}\rangle\, X_6(v,c) \vee \\ & \langle\neg\{toss\ !v\}\rangle\, X_7(v,c) \end{array}\right\}$$

(c) HMLR

$$\left\{\begin{array}{rl} X_1 = & \langle\{toss\ ?v{:}\mathsf{nat}\}\rangle\, X_5(v,n) \\ X_2 = & \mathsf{true} \\ X_5(v,c) = & \mathsf{if}\ c>0\ \mathsf{then} \\ & \quad \langle\{dice\ !1\}\rangle\, X_2 \vee \\ & \quad \langle\{toss\ !v\}\rangle\, X_5(v,c-1) \vee \\ & \quad \langle\neg\{toss\ !v\}\rangle\, X_7(v,c) \\ & \mathsf{else} \\ & \quad \langle\{dice\ !1\}\rangle\, X_2 \\ & \mathsf{end\ if} \\ X_7(v,c) = & \langle\{toss\ !v\}\rangle\, X_5(v,c-1) \vee \\ & \langle\neg\{toss\ !v\}\rangle\, X_7(v,c) \end{array}\right\}$$

(d) guarded HMLR

$$\left\{\begin{array}{rl} X_1 = & \langle\{toss\ ?v{:}\mathsf{nat}\}\rangle\, X_5(v,n) \\ X_2 = & \mathsf{true} \\ X_5(v,c) = & \mathsf{if}\ c>0\ \mathsf{then} \\ & \quad \langle\{dice\ !1\}\rangle\, X_2 \vee \\ & \quad \langle\{toss\ !v\}\rangle\, X_5(v,c-1) \vee \\ & \quad \langle\neg\{toss\ !v\}\ \wedge \\ & \quad\ \neg\{dice\ !1\}\rangle\, X_7(v,c) \\ & \mathsf{else} \\ & \quad \langle\{dice\ !1\}\rangle\, X_2 \\ & \mathsf{end\ if} \\ X_7(v,c) = & \langle\{toss\ !v\}\rangle\, X_5(v,c-1) \vee \\ & \langle\neg\{toss\ !v\}\rangle\, X_7(v,c) \end{array}\right\}$$

(e) deterministic HMLR

$$\left\{\begin{array}{rl} Z_{1,0} = & 0.5\cdot Z_{5,1}(1,n) + \\ & 0.5\cdot \boxed{Z_{5,2}(0,n)} \\ Z_{5,1}(1,c) = & \mathsf{if}\ c>0\ \mathsf{then} \\ & \quad 0.5\cdot Z_{5,3}(1,c-1) + \\ & \quad 0.5\cdot \boxed{Z_{7,4}(1,c)} \\ & \mathsf{else} \\ & \quad 0 \\ & \mathsf{end\ if} \\ Z_{5,3}(1,c) = & \mathsf{if}\ c>0\ \mathsf{then} \\ & \quad 0.5\cdot Z_{5,7}(1,c-1) + \\ & \quad 0.5\cdot Z_{7,1}(1,c) \\ & \mathsf{else} \\ & \quad 0 \\ & \mathsf{end\ if} \\ Z_{5,7}(1,c) = & \mathsf{if}\ c>0\ \mathsf{then}\ 1 \\ & \mathsf{else}\ 1\ \mathsf{end\ if} \\ Z_{7,1}(1,c) = & 0.5\cdot Z_{5,3}(1,c-1) + \\ & 0.5\cdot \boxed{Z_{7,4}(1,c)} \end{array}\right\}$$

(f) LES (partially instantiated)

$$\left\{\begin{array}{rl} Z_{1,0} = & 0.5\cdot Z_{5,1}(1,2) \\ Z_{5,1}(1,2) = & 0.5\cdot Z_{5,3}(1,1) \\ Z_{5,3}(1,1) = & 0.5 + 0.5\cdot Z_{5,3}(1,0) \\ Z_{5,3}(1,0) = & 0 \end{array}\right\}$$

(g) LES (instantiated for $n=2$)

Fig. 7: Model checking formula $\Psi_2 = \{\{toss\ ?v{:}\mathsf{nat}\}.((\neg\{toss\ !v\})^*.\{toss\ !v\})\{...\ n\}.\{dice\ !1\}\}_{\geq 0.1}$ on a PTS

Ensuring this becomes undecidable in the presence of the generalized iteration operator "loop", which enables the specification of arbitrary (Turing computable) paths in a PTS. However, the model checking procedure terminates for most practical cases of data handling regular formulas denoting finite paths (counting, accumulating or aggregating values, computing costs over paths).

5 Tool support

We implemented the on-the-fly model checking method described in Sections 3 and 4 within the CADP toolbox [27]. We briefly present here the extension of the existing on-the-fly model checker of MCL with the regular probabilistic operator, and the representations of PTS and DTMC models suitable for probabilistic analysis.

Extension of MCL and its model checker. 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. Also, using the MCL quantifiers on finite data domains, one can succinctly describe data handling probabilistic properties, such as the fact that after zero or more coin tosses, each side of the dice is reached with 1/6 probability: forall i:nat among $\{1...6\}.\{\ \{toss\ ...\}^*.\{dice\ !i\}\ \}_{\geq 1/6}$.

We also enhanced the EVALUATOR [48] 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 [43], which already serves as verification back-end for (non-probabilistic) MCL formulas. The resolution algorithms work in a systolic manner: several subsequent invocations of an algorithm for solving different variables of a BES have a linear overall complexity in the size of the BES (number of variables and operators), obtained by keeping the values of all variables computed during intermediate resolutions.

For the on-the-fly resolution of LESs, we designed a local algorithm operating on the associated Signal Flow Graphs (SFG) [12], in a way similar to the BES resolution algorithms, which operate on the associated Boolean graphs [3]. Figure 8 illustrates the SFG of the LES corresponding to the evaluation of property Ψ_1 on the PTS of the six-sided dice, shown in Figure 2(f). In the SFG, each vertex corresponds to a variable of the LES and each edge $Z_{i,s} \xrightarrow{q} Z_{j,r}$ denotes a dependency of $Z_{i,s}$ upon $Z_{j,r}$, i.e., the fact that the term $q \cdot Z_{j,r}$ occurs in the right hand side of the equation defining $Z_{i,s}$ (e.g., edges $Z_{1,0} \xrightarrow{0.5} Z_{1,1}$ and $Z_{1,0} \xrightarrow{0.5} Z_{7,2}$ represent the equation $Z_{1,0} = 0.5 \cdot Z_{1,1} + 0.5 \cdot Z_{7,2}$). Constants are represented as sink vertices (e.g., $Z_{1,1}$ or $Z_{1,10}$). Self-loops $Z_{i,s} \xrightarrow{q} Z_{i,s}$ are deleted during the construction of the SFG by updating the coefficients on the edges going out of $Z_{i,s}$ (an edge $Z_{i,s} \xrightarrow{q_{ij}} Z_{j,r}$ becomes $Z_{i,s} \xrightarrow{q_{ij}/(1-q)} Z_{j,r}$).

The on-the-fly LES resolution algorithm carries out a forward DFS (depth-first search) exploration of the SFG with detection of SCCs (strongly connected components), starting at the vertex of interest. The SFG is constructed on demand in the following way. For each vertex $Z_{i,s}$, the BES solver is invoked on the Boolean variable $X_{i,s}$ to determine whether s has an outgoing sequence matching the suffix of the regular formula β denoted by the propositional variable X_i of the determinized HMLR specification (see Fig. 2(e)). If variable $X_{i,s}$ is false, the vertex $Z_{i,s}$ denotes a constant 0, like the boxed vertices $Z_{1,1}$, $Z_{7,11}$, and $Z_{7,12}$ in Figure 8. When a constant vertex $Z_{j,r}$ is encountered, its value is propagated backwards to its predecessor vertex $Z_{i,s}$ on the DFS stack, which amounts to perform the substitution of $Z_{j,r}$ in the right-hand side of the equation defining $Z_{i,s}$. If the vertex $Z_{i,s}$ on top of the stack becomes *stable* (i.e., the values of all its successor vertices have been computed), its value is stored for possible later reuse and is propagated backwards as if $Z_{i,s}$ was a constant.

If variable $X_{i,s}$ is true, then the SFG edges going out of $Z_{i,s}$ are constructed by scanning the right hand side of the corresponding equation in the LES, and the DFS exploration can proceed further. When a non-trivial SCC (i.e., containing at least two vertices) is encountered, the corresponding LES fragment induced by the vertices of the SCC is solved using either Gaussian elimination (for LES fragments with less than $1,000$ variables) or the Gauss-Seidel iterative method (for larger LES fragments). For the SFG in Figure 8, the DFS traversal will propagate backwards the values of vertices $Z_{1,1}$, $Z_{1,10}$, and $Z_{7,11}$, stabilizing the vertex $Z_{1,5}$ at 0.5. Then, after propagating backwards the vertex $Z_{7,12}$, the SCC $\{Z_{7,6}, Z_{1,2}\}$ is detected and solved, stabilizing the vertices $Z_{7,6}$ and $Z_{1,2}$ at 1/6 and 1/3, respectively. Finally, vertex $Z_{7,6}$ is propagated and stabilizes vertex $Z_{7,2}$ at 1/3, and then the vertex of interest $Z_{1,0}$ at 1/6.

For the acyclic parts of the SFG, this algorithm amounts to a resolution by substitution, and stores in memory only the vertices (corresponding to LES variables). For each non-trivial SCC, the algorithm stores the vertices and the edges (corresponding to the operands in the right-hand sides of the LES equations) until the SCC has been solved. Therefore, the peak memory consumption is determined by the largest SCC contained in the SFG. In practice, it is often the case that the evaluation of probabilistic operators on PTSs produces LESs with SFGs that are not strongly connected (this is the case, e.g., for the SFG in Figure 8 and also for all properties shown in Section 6). In these situations, the SFG-based on-the-fly LES resolution can be

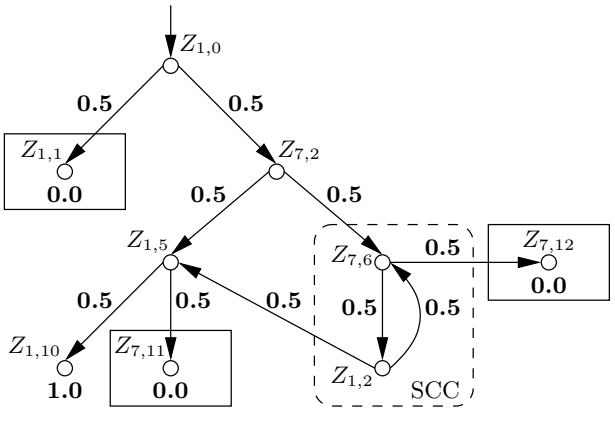


Fig. 8: Signal Flow Graph of the LES in Figure 2(f)

less memory consuming than a global resolution working on the entirely constructed LES.

Representation of PTSs. The concurrent systems under analysis in CADP are described formally using the LNT language [11], inspired from value-passing process algebras and classical programming languages. An LNT description is compiled into an LTS represented implicitly as a C program implementing the successor function according to the OPEN/CAESAR application programming interface [25] for on-the-fly graph exploration. This LTS is converted on the fly into a PTS by adding probabilities to its actions (transition labels) using the label renaming feature of EVALUATOR. By default, all transitions going out of a state are equiprobable, in which case there is no need to specify any probability distribution (this happens, e.g., for the PTSs modeling the dice, shown in Fig. 2(a) and 7(a)). If one assigns a specific probability p to some action a, then for each state s having an outgoing transition $s \xrightarrow{a} s'$, the remaining probability $1 - p$ is equally distributed on the actions labeling the neighbour transitions $s \xrightarrow{b} s'$. In this way, an LNT description can be converted on the fly (without any additional annotation) into a PTS during the verification of a probabilistic MCL formula.

Representation of DTMCs. Although the modeling and verification approach of CADP is action-based, the underlying models can be adapted to accomodate for state-based analysis, by encoding the relevant information on actions instead of states. Thus, a DTMC having atomic propositions on states can be converted into a PTS by "pushing" all state information on the actions labeling the outgoing transitions. For example, if a set of atomic propositions $q_1, ..., q_n$ hold on a state s of a DTMC, then the transitions going out of s in the corresponding PTS will be labeled by actions of the form "$A\ q_1 \dots q_n$" and will keep the same probabilities as in the DTMC. This scheme, which does not change the structure of the graph, relies on the data handling modalities of MCL for expressing atomic propositions on states: a state satisfying q_i in the DTMC satisfies the diamond modality $\langle \{A\ ?\mathsf{any}\ ...\ ?\mathsf{any}\ !q_i\ ...\} \rangle\, \mathsf{true}$ in the PTS, where "?any" is a wildcard matching a value of any type. Using the macro-definition mechanism of MCL, the atomic propositions can be encoded as macros with symbolic names and reused in state-based formulas.

In this way, one can check state-based properties using the PCTL operators encoded in terms of $\{\beta\}_{\geq p}$ as shown in Section 4.2. Moreover, the regular operators of MCL enable one to specify properties more expressive than Until operators, such as $\{send?.\mathsf{true}.(retry?.\mathsf{true}^*)^*.recv?\}_{\geq 0.9}$, where the atomic propositions like *send* are macro-definition names for the modalities $\langle \{A\ !send\} \rangle\, \mathsf{true}$ and they are checked on states using the testing operator "?" of PDL.

6 Case studies

We illustrate the application of the regular probabilistic operator by carrying out a probabilistic analysis of several concurrent systems. First, we consider several shared-memory mutual exclusion protocols (Sec. 6.1), and corroborate the results of the probabilistic analysis with those previously obtained by a steady-state analysis using CTMCs in [46]. Then, we compare our on-the-fly model checker procedure with the explicit-state algorithms of the probabilistic model checker PRISM [36], by analyzing the Bounded Retransmission Protocol (Sec. 6.2) and two randomized protocols (Sec. 6.3). All model checking experiments have been carried out on 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 [9].

We considered the explicit-state engine of PRISM in order to achieve a consistent comparison with our on-the-fly verification algorithms, which operate on explicit-state PTSs; however, some of the performance figures reported below (e.g., in Sec. 6.2) are close to those of the PRISM benchmarks[1], which were conducted using the symbolic engine of PRISM. The time and memory consumption measured for the two tools include the time of constructing the model (i.e., launching the Java virtual machine and building the DTMC by PRISM, compiling the LNT description and converting the state space on the fly into a PTS by CADP). Given that PRISM and CADP are very different in their architecture, algorithms, and implementation, the experimental measures presented in this section are not meant to provide an absolutely precise comparison of the tools performance, but rather to give general tendencies for various sizes of the models considered.

[1] http://www.prismmodelchecker.org/casestudies/

6.1 Mutual exclusion protocols

We focus here on a subset of the 27 protocols specified in LNT and studied in [46], namely the CLH, MCS, Burns&Lynch (BL), Lamport, Peterson, TAS and TTAS protocols, by considering configurations of $N \leq 5$ 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. During these sections, each protocol carries out various operations (read, write, test-and-set, etc.) on the shared variables.

In the PTS models of the protocols, all transitions going out of each state have equal probabilities, meaning that each process has the same chances to execute. This kind of analysis by introducing probabilities may seem artificial in the case of mutual exclusion protocols. However, it provides a simple way to compare the performance of different protocols by estimating their respective "speed" in reaching certain actions, and also enabling to simulate particular situations of system load (e.g., for the contention property studied at the end of this section) by varying the probabilities assigned to the actions of certain processes. We found that this probabilistic analysis corroborates, and also complements, a stochastic analysis using IMCs (*Interactive Markov Chains*) previously carried out in [46].

We formulate four probabilistic properties using MCL and evaluate them on the fly on each LNT protocol description. Since we are interested in the probability measures, we use the syntax $\{\beta\}_{\bowtie ?\, p}$ instructing the model checker to display the probability measure computed for β in addition to the Boolean verdict. 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 [26].

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:

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

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 checking the above formula for $N = 5$: for the BL protocol, which gives higher priority to processes of lower index, the probabilities computed are 71.01% for P_0, 21.33% for P_1, 5.68% for P_2, 1.53% for P_3, and 0.43% for P_4, whereas they are equal to 20% for all processes of the other, symmetric, protocols. This corroborates the results obtained by steady-state analysis using IMCs [46], which pointed out the asymmetry of the BL protocol.

The SFGs of the LESs produced by EVALUATOR when checking this formula are not strongly connected, being even acyclic as e.g., the SFG obtained for the TAS protocol.

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 [46] 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 performed by P_i depends on the placement of the memory in the hierarchy (local caches, shared RAM, remote disks) and is represented in the PTS by actions of the form "MU ... c i" [46].

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. The regular formula 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:

```
{  (¬{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 "loop" subformula denotes the entry section of P_i and requires that it terminates when the cost of all memory accesses performed by P_i (accumulated in the iteration variable *total_cost*) exceeds a given value **max**. The costs present on transitions are captured by the action pattern "{MU ... ?c:nat !i}" and used in the "continue" subformula to update the value of *total_cost*. The other processes can execute freely during the entry section of P_i, in particular they can overtake P_i by accessing their critical sections before it.

Figure 9(a) shows the probability of entering the critical section for various values of **max**. Since the entry section contains waiting loops, the 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

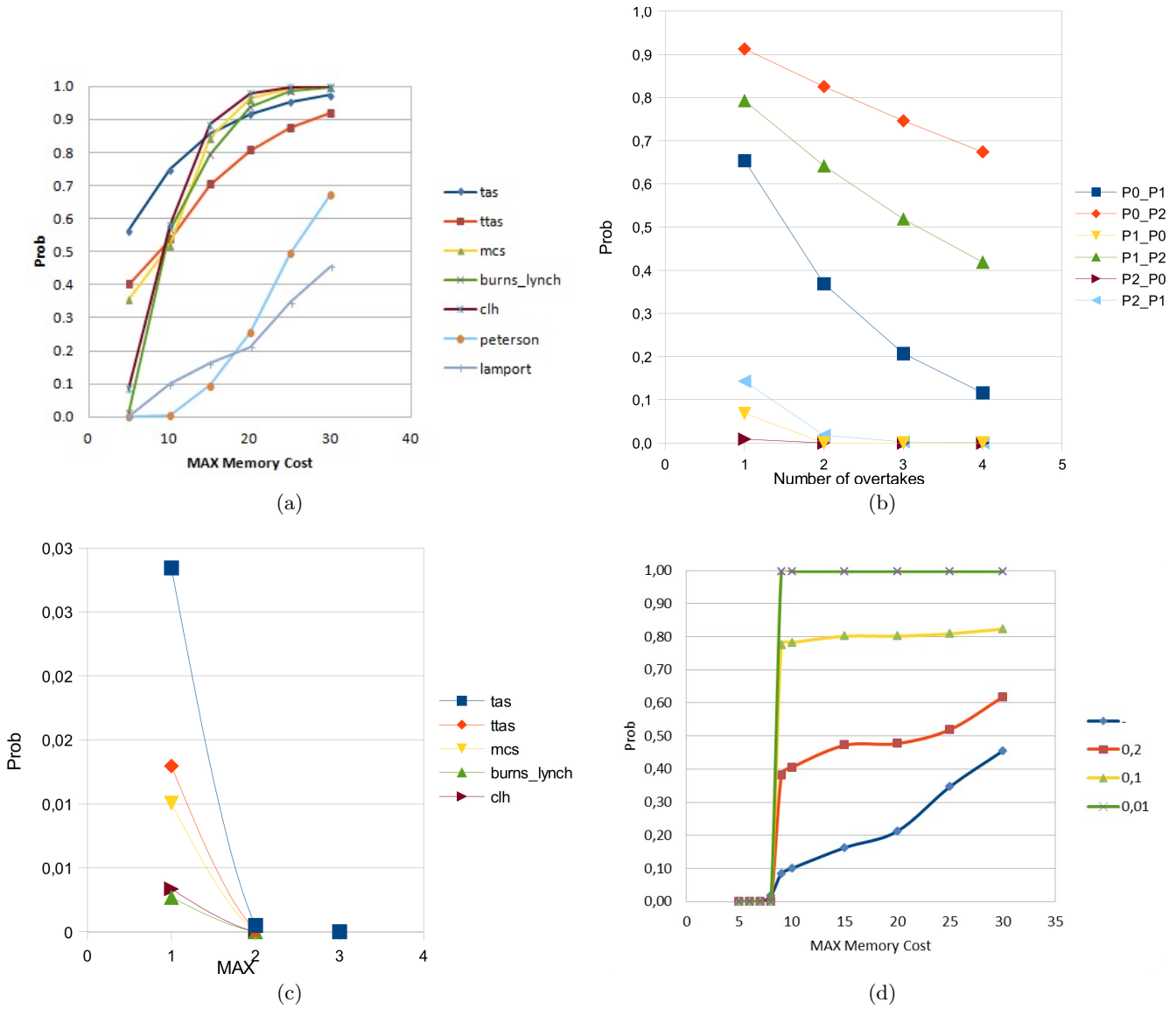


Fig. 9: Probabilities computed using on-the-fly model checking. (a) Accessing the critical section after memory accesses of cost MAX. (b) Overtaking of P_i by P_j (P_j_P_i) in the BL protocol. (c) Standalone execution of P_i. (d) Execution of P_2 (in Lamport's protocol) for different degrees of contention from the other processes.

of starvation-free protocols. This explains the asymptotic probability 1.0 observed in Figure 9(a): a process has better chances to reach its critical section when the memory cost of its entry section increases. Moreover, the variation of probabilities shown in Figure 9(a) provides a comparison of the protocols w.r.t. the "speed" of reaching the critical section: the CLH protocol has the fastest increase of the probability, and therefore it uses most efficiently its accesses to shared variables.

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%). The SFGs of the LESs produced by EVALUATOR for this MCL formula contain several SCCs, e.g., for the TAS protocol with 2 processes the SFG contains up to 32 SCCs, of which the largest one has 9 variables and was solved in 7 iterations. 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.

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 crit-

ical 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 [46] 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 9(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. The SFGs of the LESs produced by EVALUATOR contain several SCCs, e.g., for the TAS protocol with 2 processes the SFG contains up to 6 SCCs, of which the largest one has 5 variables and was solved in 7 iterations.

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 was explicitly formulated in [20] 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. The MCL formula below specifies such kind of execution, where **max** denotes the number of consecutive executions of P_i:

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

Figure 9(c) shows that the probability of standalone execution of P_i decreases with **max**, which reflects the starvation-free nature of the protocols. The SFGs of the LESs produced by EVALUATOR for this formula contain several SCCs, e.g., for the TAS protocol with 2 processes the SFG contains up to 27 SCCs, of which the largest one has 5 variables and was solved in 7 iterations.

Contention. The last property we consider regards the access of a process P_i to its critical section when the other processes attempt (with different probabilities) to access their own critical sections. Lamport's protocol [37] was designed to ensure a fast access, consisting of only a few operations on the shared variables, of a process to its critical section when the other processes do not attempt similar accesses. This was motivated by the observation that contention of accesses to critical sections is a rare phenomenon in well-designed concurrent systems. We consider the same MCL formula used for analyzing the memory latency, and evaluate it for the process P_2 in Lamport's protocol with $N = 3$ in presence of different degrees of contention, modeled by assigning on the fly different probabilities to the memory operations performed by the other processes P_0 and P_1.

The results are shown in Figure 9(d), where the curve marked with a '-' represents the normal situation (all processes compete for the critical section with equal probabilities). The plots confirm the characteristic of the fast access protocol: for low contention (1%) from the other processes, P_2 reaches its critical section with almost 100% probability after a small amount of memory operations, whereas this probability decreases for higher degrees of contention.

Performance of analysis. The sizes of the PTSs including the additional transitions of memory access costs (available in [46, Tab. 4]), 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 depend 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.

Analysis of peak cost paths. The combination of data handling and probabilistic on-the-fly model checking using MCL enables the analysis of paths having peak cost values in a PTS. Assuming that the paths of interest are characterized by a regular formula $\beta(cost)$ with a parameter *cost*, two cases must be considered:

- If the paths of interest are finite, the *peak* value of cost can be computed by checking the modality $\langle\beta(cost)\rangle\,\mathsf{true}$ with various values of *cost*, using dichotomic search to reduce the number of invocations of EVALUATOR. Then, the probability of the peak cost sequence is obtained by checking the formula $\{\beta(peak)\}_{\geq ?0}$.
- If the paths of interest are infinite, the analysis can be done by checking the formula $\{\beta(cost)\}_{\geq ?0}$ for various values of *cost* and observing the variation of the probability.

These analyses can be easily automated in CADP using SVL scripts.

6.2 Bounded Retransmission Protocol

We consider here an action-based specification of Philips' Bounded Retransmission Protocol (BRP) developed initially in LOTOS and analyzed by model checking using a data-handling version of ACTL [42]. The BRP protocol is designed to transmit large data packets over an unreliable medium by splitting them in small chunks, which are sent sequentially. After each chunk transmission, the sender waits for an acknowledgment from the receiver before sending the next chunk. If the acknowledgment does not arrive because of a communication failure, the sender times out and retransmits the chunk. In case of repeated failures, only a bounded number of retransmissions are allowed, after which the protocol aborts the transmission of the packet. The protocol informs the sender and the receiver about the outcome of each packet transmission by sending OK, NOK, or DK indications, denoting success, failure, or uncertainty (loss of the last acknowledgment) of the transmission, respectively.

To carry out a probabilistic analysis of the BRP protocol, we resorted to a state-based probabilistic modeling initially proposed in [18] and also available as a PRISM benchmark[2]. Starting from an LNT description of the BRP protocol (strongly bisimilar to the initial LOTOS description in [42]) available as a CADP demo example[3], we converted it on the fly into a PTS by setting a 2% probability for chunk losses and a 1% probability for acknowledgment losses, keeping all the other actions equiprobable, and considering the transmission of a single data packet as in [18]. We specified in MCL the action-based counterparts of the six PCTL properties given in [18], of which we illustrate below only the properties P_a and P_3, the other ones having similar shapes.

P_a evaluates the probability that the sender reports an unsuccessful transmission after the complete packet was received (this should never occur!):

$$\{\ \ (\neg\{\mathsf{OUTPUT\ !OK}\})^*.\{\mathsf{OUTPUT\ !OK}\}.\ (\neg\{\mathsf{INPUT\ !NOK}\})^*.\{\mathsf{INPUT\ !NOK}\}\ \}_{=0}$$

This MCL formula requires a zero probability measure for the sequences containing a successful indication delivered to the receiver (action "OUTPUT !OK") followed by an unsuccessful indication to the sender (action "INPUT !NOK").

P_3 evaluates the probability that the sender reports an unsuccessful transmission after more than eight chunks of the packet have been transmitted:

$$\{\ \ ((\neg\mathsf{REC_L})^*.\mathsf{REC_L})\ \{8\ ...\}.\ (\neg(\mathsf{REC_L} \vee \{\mathsf{INPUT\ !NOK}\}))^*.\{\mathsf{INPUT\ !NOK}\}\ \}_{\geq ?0}$$

This MCL formula asks for the probability measure of a sequence containing eight chunk acknowledgments received by the sender (action "REC_L") followed by an unsuccessful indication to the sender (action "INPUT !NOK").

We checked the six properties using explicit-state PRISM and EVALUATOR on the corresponding BRP models for five retransmissions and packet lengths between 16 and 256 chunks. The probability values for all six properties obtained using CADP and PRISM are identical to the ones reported in [18], which suggests that the action-based model in [42] and the state-based model in [18] are consistent, even if they are quite different in structure. The performance measures of CADP and PRISM are shown in Figure 10. We observe that CADP consumes up to one order of magnitude less memory than PRISM, which is in turn faster on small configurations but becomes slower for properties P_1, P_2, and P_3 on larger configurations (packets with more than 150 chunks). For all six MCL formulas, the LESs produced by EVALUATOR have acyclic SFGs.

Probability of successful packet transmission. In addition to the six probabilistic properties of the BRP analyzed in [18], we also studied how the total number of retries during a packet transmission influences the probability of the packet being successfully received. The MCL formula below computes the probability of a successful packet transmission (initiated by an action "INPUT_PACKET" and concluded by an action "INPUT !OK") when the total number of retries occurring when sending the chunks is equal to **retries**.

```
{ INPUT_PACKET.
  loop (r:nat:=0) in
    if (r = retries) then
      exit
    else
      (¬{SEND_K ...})*.{SEND_K ... ?c:nat}.
      loop (nc:nat:=0) : (rc:nat) in
        exit (nc)
        |
        (
          (¬({OUTPUT !c ?any} ∨ {SEND_K ... !c}))*.
          {SEND_K ... !c}.continue (nc + 1)
        )
      end loop.
      (¬({OUTPUT !c ?any} ∨ {SEND_K ... !c}))*.
      {OUTPUT !c ?any}.continue (r + rc)
    end if
  end loop.
  (¬{INPUT !OK})*.{INPUT !OK}
}≥?0
```

The outer "loop" subformula counts the total number of retries in the iteration variable r by summing up the retries that occur for every chunk c, i.e., the number of retransmissions (actions "SEND_K ... !c") occurring after

[2] http://www.prismmodelchecker.org/casestudies/brp.php
[3] http://cadp.inria.fr/demos.html, demo 16

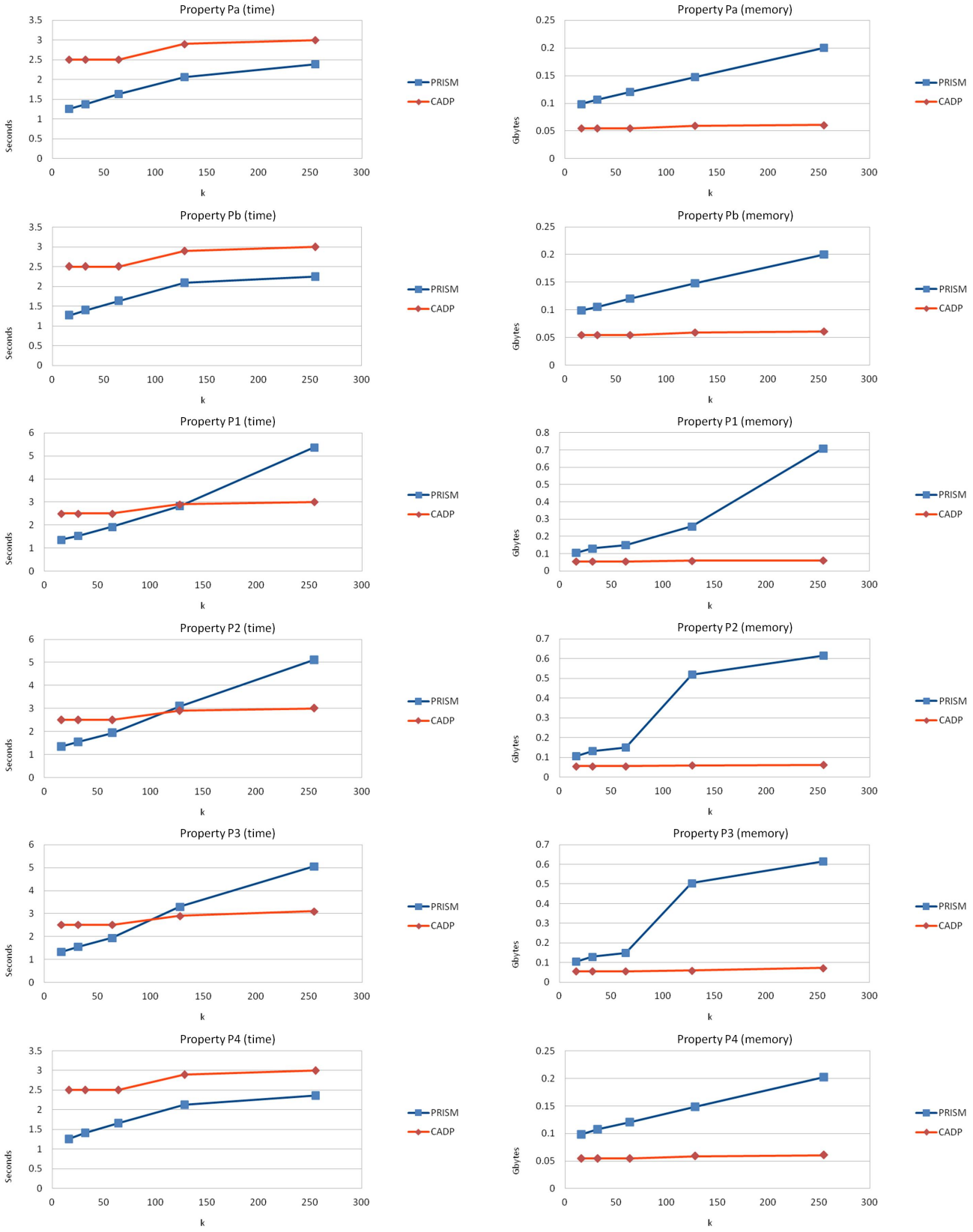


Fig. 10: Execution time and memory consumption of CADP and PRISM for checking the six probabilistic properties in [18] on the BRP model for five retries and a packet size k between 16 and 256 data chunks.

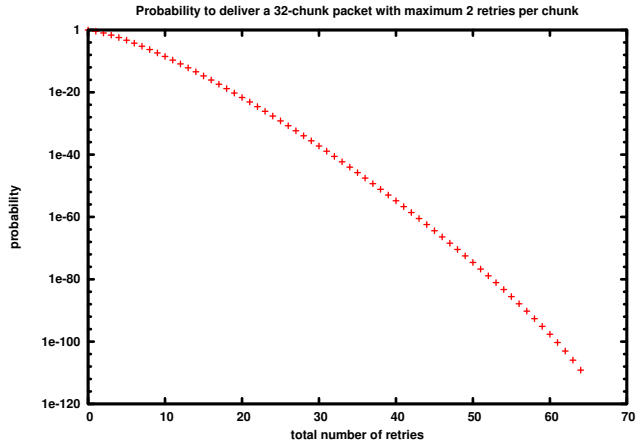


Fig. 11: Probability of successful packet transmission for the BRP model with two retries and packet size $k = 32$, depending on the total number of retries

the chunk was sent for the first time and before it was received (action "OUTPUT !c ?any"). These retries are counted by the inner "loop" in the iteration variable nc and provided as result in the return variable rc.

Figure 11 shows the variation of the probability of successfully transmitting a packet of length $k = 32$ with maximum two retries per chunk, when the total number of retries varies between 0 and 64. This probability is high ($99,9\%$) when no retries occur, and decreases very fast with the total number of retries ($0,05\%$ for five retries). This indicates that with a 2% chunk loss probability and a 1% acknowledgment loss probability, the BRP protocol is already quasi-reliable, the probability of a successful packet transmission such that the retransmission bound was reached for each chunk being negligible (10^{-110} for 64 retries).

Of course, this analysis can be carried out with PRISM as well, by introducing an additional variable in the PRISM model of the BRP for counting the total number of retries; on the other hand, the action-based analysis using the MCL formula above (which is independent of the packet length and retransmission bound parameters of the BRP) does not require to change the PTS model.

6.3 Randomized Protocols

We consider here two randomized protocols taken from the PRISM benchmarks suite. The first one is the Israeli-Jalfon (IJ) self-stabilizing protocol [32], which operates on a set of k concurrent processes P_0, ..., P_{k-1} connected on a ring topology and brings them after a finite number of steps to a stable configuration, where only one process is privileged (i.e., has a token). The IJ protocol is asynchronous and communication on the ring is bidirectional. When a process has a token, it can be scheduled for execution and it must pass the token with equal probability to its left or right neighbor process on the ring. If two tokens reach the same process, they are merged into a single one.

The fundamental property of self-stabilisation protocols is that starting from any configuration, a stable configuration is reached with probability one. This can be expressed in MCL as follows:

$$\{\ \ (\neg\varphi_{stable}?\ .\ \mathsf{true})^*.\ \varphi_{stable}?\ \ \}_{=1}$$

where

$$\varphi_{stable} = \mathsf{exists}\ i{:}\mathsf{nat}\ \mathsf{among}\ \{0\ ...\ n-1\}\ .\ (\langle\{\mathsf{EXE}\ !i\}\rangle\,\mathsf{true} \wedge [\{\mathsf{EXE}\ ?j{:}\mathsf{Nat}\ \mathsf{where}\ j \neq i\}]\,\mathsf{false})$$

The state formula φ_{stable} expresses that a single process P_i can execute an action in the current state (action "{EXE !i}").

Figure 12 shows the time and memory consumed by EVALUATOR and explicit-state PRISM for checking the self-stabilisation property on the corresponding LNT and DTMC descriptions of the IJ protocol. Both tools exhibit similar performance on medium-sized configurations ($k \leq 15$ processes), PRISM becoming more efficient for larger configurations. The SFGs of the LESs produced by EVALUATOR contain several SCCs, e.g., for $k = 10$ the SFG contains 8 SCCs, of which the largest one has 756 variables and was solved in 22 iterations.

The second example considered is a version of the randomized Lehmann-Rabin (LR) dining philosophers protocol without fairness assumptions [21]. Each of the k philosophers picks his forks in a random order if they are both available, and releases them in any order after eating and before returning to thinking. We modeled the system in LNT and checked the basic property stating that a hungry philosopher will be able to eat with probability one:

$$[\mathsf{true}^*.\{\mathsf{THINK}\ ?p{:}\mathsf{nat}\}]\ \{\ \ (\neg\{\mathsf{EAT}\ !p\})^*.\{\mathsf{EAT}\ !p\}\ \ \}_{=1}$$

This MCL formula combines a regular box modality searching all states in which a philosopher p becomes hungry (i.e., immediately after performing an action "{THINK !p}"), and a regular probabilistic operator checking the reachability of an action "{EAT !p}" with probability one. The value extraction mechanism, which captures the value of p present on a transition label in the box modality and reuses it in the probabilistic operator, enables to check configurations of the system with different values of k without changing the formula.

Figure 13 shows the performance of CADP and explicit-state PRISM for checking the above property on their respective models of the LR protocol. On this example, PRISM is increasingly faster but also increasingly demanding in memory (one order of magnitude difference with CADP for six philosophers). Also on this example, the SFGs of the LESs produced by EVALUATOR contain several SCCs, e.g., for $k = 4$ the SFG contains 180 SCCs, of which the largest one has 3016 variables and was solved in 25 iterations.

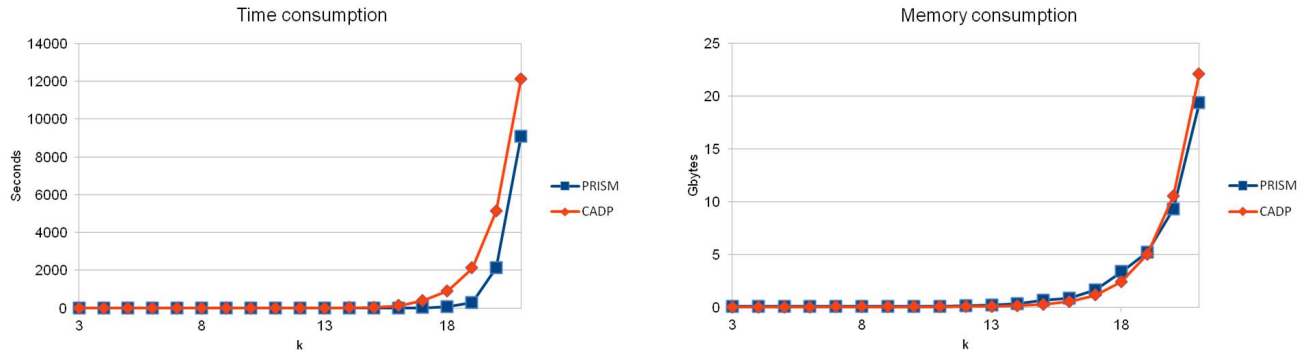


Fig. 12: Execution time and memory consumption of CADP and PRISM for computing the probability of reaching a stable state on the self-stabilising protocol with $3 \leq k \leq 20$ concurrent processes.

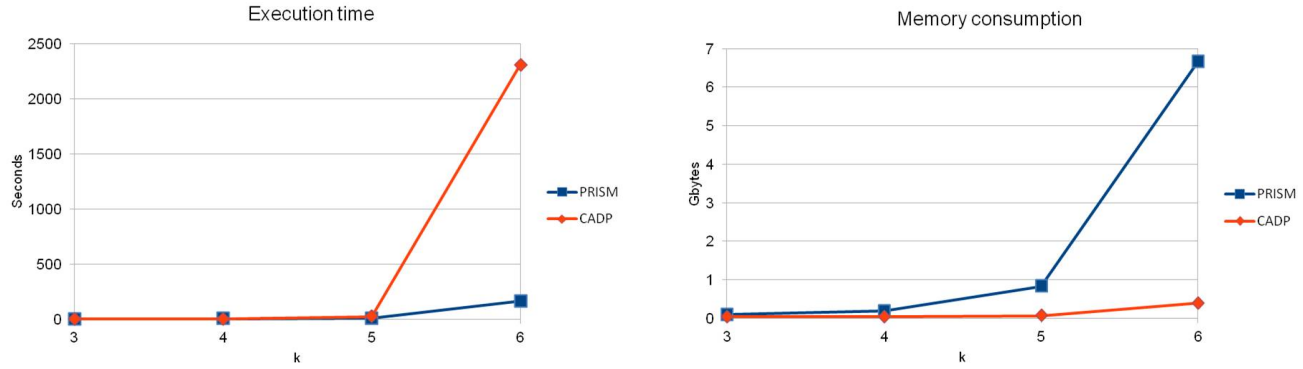


Fig. 13: Execution time and memory consumption of CADP and PRISM for computing the probability that a hungry philosopher eats on the LR dining philosophers protocol with $3 \leq k \leq 6$ philosophers.

7 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 [48] that we extended with a general iteration operator "loop" enabling the specification of complex (Turing computable) 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. Several experiments showed that this on-the-fly verification procedure exhibits a performance comparable to the explicit-state algorithms of the PRISM model checker.

Since the PTS is explored on the fly by annotating the transitions of the original LTS with probabilities, the high-level description of the concurrent system under analysis (written, in our case, in the LNT language) remains unchanged, and can serve therefore as unique model for both functional and quantitative analysis, the details of which can be encapsulated in SVL scripts.

The on-the-fly analysis back-end (which is currently sequential) can be enhanced with distributed capabilities by connecting it with the MUMPS distributed solver [2] for sparse LESs. We also 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 [17]. Finally, we will seek to extend our approach (which deals only with *finite* paths described using data handling regular formulas) to handle infinite paths satisfying ω-regular properties, along the lines of [7, Chap. 10].

Acknowledgments

This work was partially supported by the European project SENSATION (*Self Energy-Supporting Autonomous Computation*) FP7-318490. We are grateful to the anonymous referees for their constructive criticism and their valuable suggestions for improving the

manuscript. We also thank Wendelin Serwe for providing the LNT descriptions of the mutual exclusion protocols.

References

1. G. Agha, J. Meseguer, and K. Sen. PMaude: rewrite-based specification language for probabilistic object systems. *Electron. Notes Theor. Comput. Sci.*, 153(2):213–239, May 2006.
2. 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.
3. H. R. Andersen. Model checking and boolean graphs. *TCS*, 126(1):3–30, 1994.
4. A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time Markov chains. In *CAV'96*, LNCS vol. 1102, p. 269–276. Springer, 1996.
5. C. Baier, L. Cloth, B. R. Haverkort, M. Kuntz, and M. Siegle. Model checking Markov chains with actions and state labels. IEEE Trans. Software Eng., 33(4):209–224, 2007.
6. C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model-checking algorithms for continuous-time Markov chains. *IEEE Trans. Software Eng.*, 29(7):1–18, July 2003.
7. C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.
8. H. Barringer, A. Goldberg, K. Havelund, K. Sen. Rule-based runtime verification. In *VMCAI'04*, LNCS vol. 2937, p. 44-57. Springer, 2004.
9. 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.
10. J. A. Brzozowski. Derivatives of regular expressions. *JACM*, 11(4):481–494, 1964.
11. 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.
12. L. O. Chua and P. M. Lin. *Computer Aided Analysis of Electronic Circuits*. Prentice Hall, 1975.
13. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
14. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
15. R. Cleaveland, S. P. Iyer, and M. Narasimha. Probabilistic temporal logics via the modal mu-calculus. *TCS*, 342(2-3):316–350, 2005.
16. R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *FMSD*, 2(2):121–147, 1993.
17. 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.
18. P. D'Argenio, B. Jeannet, H. Jensen, and K. Larsen. Reachability analysis of probabilistic systems by successive refinements. In *PAPM/PROBMIV'01*, LNCS vol. 2165, p. 39–56. Springer, 2001.
19. N. Dershowitz. Termination of rewriting. *J. of Symb. Comput.*, 3(1):69–115, 1987.
20. E. W. Dijkstra. Solution of a problem in concurrent programming control. *CACM*, 8(9):569, 1965.
21. M. Duflot, L. Fribourg, and C. Picaronny. Randomized dining philosophers without fairness assumption. *Distributed Computing*, 17(1):65–76, 2004.
22. E. A. Emerson and C-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *LICS'86*, p. 267–278. IEEE Press, 1986.
23. W. Feller. An Introduction to Probability Theory and Its Applications. John Wiley & Sons, 2001.
24. M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *JCSS*, 18(2):194–211, 1979.
25. H. Garavel. OPEN/CAESAR: An Open Software Architecture for Verification, Simulation, and Testing. In *TACAS'98*, LNCS vol. 1384, p. 68–84. Springer, 1998. Full version available as INRIA Research Report RR-3352.
26. H. Garavel and F. Lang. SVL: a Scripting Language for Compositional Verification. In *FORTE'01*, p. 377–392. Kluwer, 2001.
27. 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.
28. R. J. van Glabbeek, S. A. Smolka, and B. Steffen. Reactive, generative and stratified models of probabilistic processes. *Inf. Comput.*, 121(1):59–80, 1995.
29. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Asp. Comput.*, 6(5):512–535, 1994.
30. M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *Journal of the ACM*, 32:137–161, 1985.
31. H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. Towards model checking stochastic process algebra. In *IFM'00*, LNCS vol. 1945, p. 420–439. Springer, 2000.
32. A. Israeli and M. Jalfon. Token management schemes and random walks yield self-stabilizating mutual exclusion. In *PODC'90*, p. 119–131. ACM Press, 1990.
33. D. E. Knuth and A. C. Yao. The complexity of nonuniform random number generation. In *Algorithms and Complexity: New Directions and Recent Results*, p. 357–428. Academic Press, New York, 1976.
34. D. Kozen. Results on the propositional μ-calculus. *TCS*, 27:333–354, 1983.
35. D. Kozen. A probabilistic PDL. *JCSS*, 30(2):162–178, 1985.
36. 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.
37. L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1-11, 1987.

38. K. G. Larsen. Proof systems for Hennessy-Milner logic with recursion. In *CAAP'88*, LNCS vol. 299, p. 215–230. Springer, 1988.
39. K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Inf. and Comput.*, 94(1):1–28, 1991.
40. D. Latella, M. Loreti, and M. Massink. On-the-fly fast mean-field model-checking. In *Trustworthy Global Computing*, p. 297–314. Springer, 2014.
41. Z. Manna and A. Pnueli. The temporal logic of reactive and concurrent systems, volume I (Specification). Springer Verlag, 1992.
42. R. Mateescu. Formal description and analysis of a bounded retransmission protocol. In *COST'96*, University of Maribor, Slovenia, 1996. Also available as INRIA Research Report RR-2965.
43. R. Mateescu. Caesar_solve: A generic library for on-the-fly resolution of alternation-free boolean equation systems. *STTT*, 8(1):37–56, 2006.
44. 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.
45. R. Mateescu and J. I. Requeno. On-the-fly model checking for extended action-based probabilistic operators. In *SPIN'2016*, LNCS vol. 9641, p. 189–207. Springer, 2016.
46. 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.
47. R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *SCP*, 46(3):255–281, 2003.
48. 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.
49. 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.
50. P. Wolper. Temporal logic can be more expressive. In *FOCS'81*, p. 340–348. IEEE Press, 1981.