Simulation-based Verification of Hybrid Automata Stochastic Logic Formulas for Stochastic Symmetric Nets

Elvio Gilberto Amparore
Università di Torino,
Dipartimento di Informatica
Torino, Italy
amparore@di.unito.it

Benoît Barbot
ENS Cachan, CNRS & INRIA
LSV
Cachan, France
barbot@lsv.ens-cachan.fr

Marco Beccuti
Università di Torino,
Dipartimento di Informatica
Torino, Italy
beccuti@di.unito.it

Susanna Donatelli
Università di Torino,
Dipartimento di Informatica
Torino, Italy
susi@di.unito.it

Giuliana Franceschinis
Università Piemonte Orientale,
DiSIT
Alessandria, Italy
giuliana.franceschinis@di.unipmn.it

ABSTRACT

The Hybrid Automata Stochastic Logic (HASL) has been recently defined as a flexible way to express classical performance measures as well as more complex, path-based ones (generically called "HASL formulas"). The considered paths are executions of Generalized Stochastic Petri Nets (GSPN), which are an extension of the basic Petri net formalism to define discrete event stochastic processes. The computation of the HASL formulas for a GSPN model is demanded to the COSMOS tool, that applies simulation techniques to the formula computation. Stochastic Symmetric Nets (SSN) are a high level Petri net formalism, of the *colored* type, in which tokens can have an identity, and it is well known that colored Petri nets allow one to describe systems in a more compact and parametric form than basic (uncolored) Petri nets. In this paper we propose to extend HASL and COSMOS to support colors, so that performance formulas for SSN can be easily defined and evaluated. This requires a new definition of the logic, to ensure that colors are taken into account in a correct and useful manner, and a significant extension of the COSMOS tool.

Keywords

Generalized Stochastic Petri Nets, Hybrid Automata Stochastic Logic, Simulation-based Verification, Stochastic Symmetric Nets

Categories and Subject Descriptors

I.6 [**Computing Methodologies**]: Simulation and Modeling

SIGSIM-PADS'13, May 19–22, 2013, Montreal, Quebec, Canada.

General Terms

Performance, Measurement

1. INTRODUCTION

Simulation of Discrete Event Stochastic Processes (DESP) is one of the most widespread techniques for evaluation of several types of quantitative (e.g. performance or reliability) measures. Although in a number of cases it can be combined with analytical or numerical analysis techniques, stochastic simulation is the preferred technique when the system is very complex and large, and/or its state space is very large or potentially infinite, or when some of the system characteristics (e.g. delay probability distributions, complex scheduling policies) prevent the application of other mathematical methods.

On the other hand, a class of performance measures that has drawn significant attention in the last years is that of path-based formulas: formulas that take into account a subset of the behaviors of the system. Path formulas may prove useful, for example, to compute the probability of the set of paths that lead to a safe final state in a reliability model, or the time to complete a production in a flexible manufacturing system when no machine breaks down during the production phase. Examples of path-based languages are the Extended Stochastic Probes (XSP [10], that define passage time over a subset of the behaviors of a PEPA model [14]), or the formalism of Path Automata (PA, operating on Stochastic Activity Networks [16]). Both XSP and PA have been devised as an extension of classical performance evaluation measures.

Another set of languages that allows to "deal with paths" comes from the community of temporal logics (logics that define path-based qualitative properties, like CTL [12] or LTL [17]) and of the associated stochastic extension, like Continuous Stochastic Logic (CSL)[1] and CSLTA [11, 8]. Here the paths considered are timed executions of a Continuous Time Markov Chain (CTMC). Paths in CSL are identified through two path operators, the time-bounded neXt and the time-bounded Until, while, to allow more flexibility, CSLTA identifies paths through a timed automata that "reads" timed execution traces. As usual in stochastic model checking the "target measure" is the computation of

the probability of the set of timed paths that satisfies the path formula or, for CSLTA, the probability of the set of timed paths that are accepted by the timed automata. To be precise the logic formula being evaluated is the following "is the probability of accepted paths greater or equal (or less than) a given threshold?", which obviously requires the computation of the path probabilities. For the desire to stay within analytical solution methods CSL and CSLTA only deal with CTMCs, although the size of the models has led to the use of statistical model checkers, that compute path probabilities via discrete event simulation. Examples of statistical model checking tools for CSL are: YMER [21] and VESTA [18]. VESTA is a Java-based tool for statistical analysis of probabilistic systems. It implements statistical methods based on Monte-Carlo simulation and statistical hypothesis testing. YMER is instead a command-line tool, written in C/C++, for verifying transient properties of CTMCs and generalizations. It implements statistical CSL model checking techniques, based on discrete event simulation and acceptance sampling.

A language that subsumes path-based performance measures as defined in Path Automata or in XSP, as well as the stochastic logics of CSL and CSLTA is the one defined for the Hybrid Automata Stochastic Logic (HASL) which has been recently introduced in [5]. HASL addresses Generalized Stochastic Petri Nets (GSPN) [15] models with general distributions (thus going beyond the CTMC limits) and defines paths through the Linear Hybrid Automata (LHA) that allows a richer definition of paths than CSL and CSLTA. The COSMOS tool [4] has been developed to evaluate HASL formulas on GSPNs. COSMOS evaluates the HASL formulas using simulation techniques, it is therefore considered a tool in the family of statistical model checkers.

This paper proposes an extension of HASL to deal with a colored extension of GSPN known as Stochastic Symmetric Nets (SSN) [9]. The HASL language has been redefined to work with the "colored" tokens and transition instances of SSN, leading to a more parametric and compact language for large systems with symmetric structure and behavior. The COSMOS tool has been extended consequently.

The expressiveness of SSNs and of the extension of HASL to handle *colors*, allows to build compact models (and compact and parametric formulas) for complex systems comprising several similarly behaving components. Colors in SSNs allow to both represent (discrete) data structures and to uniquely identify "customers" or "resources", a feature that is particularly useful when defining path properties that refer to a sequence of steps through which a given customer or resource must undergo (e.g. average processing time of a part entering a pipeline, conditioned on the occurrence of a limited number of breakdowns along the path).

The paper is organized as follows: Section 2 introduces background material on the GSPN formalism and on HASL, and introduces a number of small LHA to exemplify the specification of basic performance indices.The COSMOS tool is also briefly reviewed. Section 3 presents SSN and the extension of HASL to account for colors. The extension proposed is the discussed and evaluated on three main examples in Section 4. Planned future works are discussed in the last section, concluding the paper.

2. BACKGROUND: THE GSPN FORMALISM AND HASL

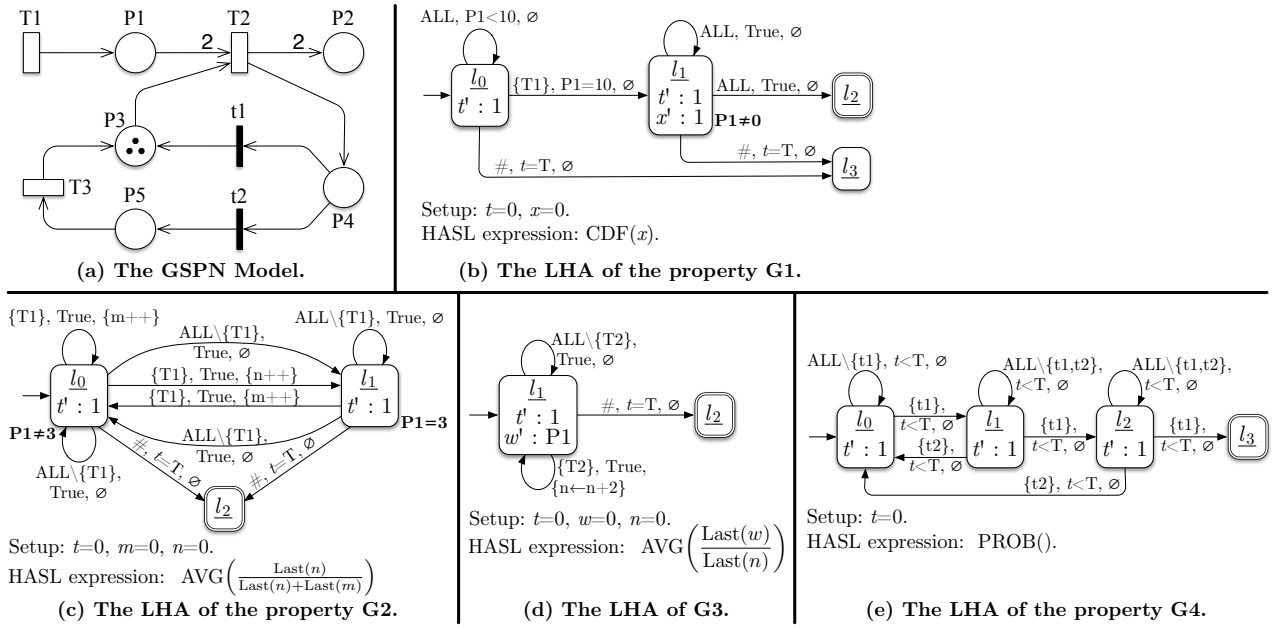
In this section we shall present the background material, in particular the formalism of GSPN and the logic HASL. We shall also briefly recall the main features of the tool COSMOS, that evaluates HASL properties over GSPN models.

2.1 The GSPN formalism

Generalized Stochastic Petri Nets[15] (GSPN) are a formalism to describe discrete event dynamic systems. They have widely been used in the past to describe and evaluate systems, and we recall them briefly in the following. A GSPN is a bipartite directed graph with weighted arcs in which nodes are split into places (circles) and transitions (bars). Each place may contain zero or more tokens, and the state of the GSPN is determined by the number of tokens in each place. The state change is an effect of the firing of a transition: the direct arcs connecting places to transitions define the conditions under which the transition may fire (enabling) while the whole set of arcs to and from a transition defines the state change induced by the firing of the transition. Figure 1(a), top, shows a simple example of a GSPN model of a processor sharing open queue in which jobs are served at batches of 2, and there are 3 servers attending the queue, were each server may go idle for a while after having provided service. Transition T_1 describes the arrival process, T_2 the service time, T_3 the idle time, and t_1 and t_2 the decision of going idle (t_2) or not (t_1).

The enabling of a transition t requires that there are at least n_p tokens in each input place p of t, where n_p is the weight of the arc from p to t. If no weight is indicated, the weight is 1 by default. The firing of t removes n_p tokens from each input place p and adds $n_{p'}$ tokens to each output place p'. For the GSPN in the figure, $T1$ is always enabled, T_2 requires two jobs in the queue (place P_1) and a server available (place P_3), and its firing removes two tokens from P_1 and one from P_3, while adding two tokens into the output buffer (place P_2) and a token into the server decision place P_4. From P_4 the server may either choose to go idle (place P_5) or to go directly back to work (place P_3).

Box transitions describe a timed event with an associated delay probability distribution, the exponential distribution being the default, while thin transitions describe events that require no time (typically a decision, as in the case of t_1 and t_2). A GSPN describes therefore a stochastic process, and if all transitions are exponential or immediate this process is a CTMC, and analytical techniques can be applied to compute performance indices. When either the discrete state space is potentially infinite (as in our example) or the transitions are not exponential, simulation becomes the most widespread analysis technique. Classical performance indices includes the distribution (or mean value) of tokens in a place and the throughput of a transition. More elaborate performance indices, like the distribution of the passage time over a subnet, may not be directly computed upon the basic performance indicators [2] hence more elaborate ways of defining performance indices for a GSPN are needed. In this paper we concentrate on the computation of performance indices based on timed paths, where a path is a net execution (an (in)finite sequence of transitions and associated delays), in other words, a simulation trace of the GSPN. With reference to the GSPN of Figure 1, we might be interested in the probability that a customer, that arrives in the queue



(a) The GSPN Model. **(b) The LHA of the property G1.**

(c) The LHA of the property G2. **(d) The LHA of G3.** **(e) The LHA of the property G4.**

Figure 1: A simple GSPN model and a set of properties written as LHAs.

P_1 while there are already k tokens in there, experiences a time to reach the output buffer P_2 greater than a threshold value T_{max}.

It has been already observed by many authors (for example [11, 16]) that automata can be a good way to specify properties that depend on a subset of the possible executions, since an automaton defines the language of "accepted paths" (paths of the GSPN recognized by the automaton), and that, in particular, timed automata can be used when the executions to be considered include some timing constraints. As explained in the introduction, the performance indices we consider are the ones that can be specified using the HASL language.

2.2 Hybrid Automata Stochastic Logic

A specification in HASL is composed of a Linear Hybrid Automaton (LHA) and one or more formulas for the performance indicators. LHA are automata (composed of a set of locations and a set of edges) enriched with a set of variables. The state of an LHA is therefore a pair (location, variables' evaluation). The purpose of an LHA is to recognize (accept or reject) simulation traces of a GSPN (also called timed executions in the model checking community) and perform measures on the accepted paths by properly updating its variables. Figure 1(b) is an example of LHA with four locations: with the common graphical notation, l_0 is identified as an initial location (incoming arrow) and l_2 as a final one (double border). LHA variables come in two flavors: stable variables, which are discrete variables that can be updated only when an edge is taken, and flow variables, which are continuous variables that are continuously increased or decreased linearly according to a derivative that is specified in each location of the automaton. Also edges come in two flavors: synchronized edges that are taken when one of the GSPN transitions in its "synchronization set" fires in the simulation trace, and autonomous edges that can be taken by the LHA as soon as a condition on the LHA variables and/or the GSPN marking evaluates to true. Every edge may have an associated condition on the variables and on the marking (for the edge to be taken) and an update of the LHA variables (to be executed when the edge is taken). Autonomous edges are taken as soon as the associated condition is verified, with priority over other arcs.

An example of synchronized edge in the LHA of Figure 1(b) is that from l_0 to l_1, which synchronizes with the firing of transition T_1 in the trace under the condition that the GSPN marking reached after firing T_1 has 10 tokens in place P_1; this edge does not update any variable. An example of autonomous edge is that from l_0 to l_3; The sharp sign indicates that the edge is of the autonomous type, and condition $t = T$ indicates that the edge can be taken as soon as the LHA flow variable t is equal to the constant value T. Locations can have an associated condition, that may involve both the stable variables of the LHA and the marking of the GSPN. An example is shown in the LHA of Figure 1(c): location l_1 has an associated condition $P_1 = 3$, meaning that, while recognizing a GSPN trace, the LHA can stay in location l_1, or can reach this location, only if the marking reached in the trace has 3 tokens in place P_1.

Formulas in HASL allow one to define the following performance indicators to be computed on the observed traces: the probability that a GSPN trace lead the automaton to a final state, i.e. that the trace is accepted by the automaton (operator PROB), the cumulative distribution of a trace expression (operator CDF), the average over all accepted traces of a trace expression (operator AVG). If ve is a variable expression (arithmetic expression of the value of LHA variables and of the number of tokens in the net places), then a trace expression has the form last(ve), min(ve), max(ve), mean(ve), integral(ve) which define, for a given trace, respectively the last value of ve, the min/max value of ve, the time-weighted average of ve, and the integral of ve.

The LHA describes which simulation traces are of interest: while simulating a GSPN execution trace, each transition

firing of the GSPN has to be matched by a corresponding synchronized edge, otherwise the trace is discarded. In addition, autonomous edges provide an "internal" behavior of the LHA that is independent from the GSPN model execution (e.g. to represent timeouts or to detect a given condition). Hence, the GSPN and LHA evolve in parallel until the LHA reaches either an accepting final state or a state where there are no autonomous edges enabled and no edges that can synchronize with the GSPN next transition (in the former case the trace is accepted, in the latter it is immediately discarded). Along the trace, the LHS cannot influence the GSPN behavior; note that one has to specify self-loops labeled with all GSPN transitions that do not cause a LHA location change, but should not cause the trace to be discarded (e.g. the self loop on l_1 in Figure 1(c) with synchronization set ALL$\backslash\{T_1\}$ is needed, otherwise the path would be rejected if a transition different from T_1 were fired in the GSPN).

The HASL measures are computed only from the set of accepted traces (i.e. execution paths that reach a final state in the LHA). When the LHA moves to a final location, the simulation ends, and the HASL measures are updated. Simulations are run (in parallel on multi-core systems) until all the estimators reach the specified accuracy. The measure PROB gives the fraction of simulations accepted by the LHA.

To better illustrate the use of HASL for the definition of performance indices, we shall consider a few examples. The reader may refer to [5] for a formal definition of HASL.

Figure 1(c) is an LHA, whose objective is to compute, for the time interval $[0, T]$, the probability that an arriving customer (token increase in place P_1 due to the firing of T_1) "sees" 2 customers already in P_1 (property **G2**). The LHA has two stable, discrete variables, n and m, to account for the number of arrivals that see (or do not see) two tokens in P_1, and a flow variable t, that is continuously and linearly incremented with derivative 1 in both locations: variable t plays the role of a timer that counts the time elapsed in the path being recognized by the LHA. There are two non-final locations, characterized by the conditions $P_1 \neq 3$ (location l_0) and $P_1 = 3$ (location l_1) on the marking of the GSPN, and a final one (location l_2, identified by a double border) that is reached when the time elapsed along the considered path is greater than a threshold value T.

For a path that starts in the initial marking of the GSPN, the automaton stays in location l_0, accepting all transitions. When the transition that fires is T_1 the behavior of the automaton depends on the marking of the net *after* the firing of T_1: if the number of tokens is different from 3 it stays in l_0 and increments the variable m, if it is equal to 3 it increments variable n, since this firing of T_1 has taken place when there were 2 customers in P_1. When variable t reaches the threshold T, the automaton moves to the final location l_2. Note that the automaton is deterministic: given a marking and a transition there is a single arc of the automaton that can be taken. This is a specific requirement of HASL, otherwise the resulting stochastic process would be partially undefined (or better, underspecified). The formula computed is AVG(last(n)/(last(n)+last(m)), which gives precisely the estimation of the probability that an arriving customer sees exactly two tokens in the queue.

The LHA of Figure 1(d) allows one to compute the average waiting time in $P1$, in the $[0, T]$ interval (property **G3**). The waiting time is computed as the total of the waiting times experienced by each single token in P_1 (accumulated in the LHA variable w), divided by the total number of customers that have left the system (accumulated in the discrete variable n). Note that n is incremented by 2 at each firing of T_2, due to the batch service type, while variable w is used to accumulate the time spent in P_1, and it is linearly incremented as time elapses, with a derivative equal to the number of tokens in P_1. The waiting time is then estimated by AVG(last(w)/(last(n)). Using variable w we can also compute the mean number of tokens in P_1, as AVG(last(w)/(last(t)) (considering that t counts the time elapsed along the accepted trace, this formula can also be expressed using the timed weighted average mean, as AVG(mean(w)).

The LHA of Figure 1(b) shows instead an example of computation of the distribution of the time it takes to empty a queue starting from an arrival that "sees" 9 tokens upon arrival, on the interval $[0, T]$ (property **G1**). The LHA uses two flow variables: t to accumulate the elapsed time along the path, as in the previous examples, and x to accumulate the time from when the queue goes to 10 to when it goes to zero. Note the use of two locations without any outgoing arc: l_2 is final (the corresponding path will be taken into account in the performance computation), while l_3 is not (and therefore the paths ending up in l_3 will not be considered). Location l_0 is used to accept all the traces until there is a firing of T_1 that leads to a queue (P_1) of 10 customers, which takes the automaton to location l_1. Note that x is incremented only when the automaton is in location l_1. The CDF is then estimated using the formula CDF(last(x)).

The last LHA considered, depicted in Figure 1(e), shows an example of the use of the automaton to recognize paths with specific conditions upon the transitions that fire along the path. The performance index considered is the probability of the executions that, up to time T, encounter three consecutive services in which the server does not go idle after having provided service (property **G4**). A server goes idle when transition t_2 is executed, while it goes immediately back to service when t_1 is executed, therefore the automaton accepts only paths that see 3 firing of t_1 without any firing in t_2 in between. The performance indicator is then PROB() which "counts" the percentage of GSPN executions that are accepted by the automaton.

2.3 The COSMOS tool

The performance indicators specified in HASL may be very difficult to compute analytically, and indeed the most promising performance evaluation technique in this case is simulation. The COSMOS simulator [4] computes the estimation of the PROB(), CDF(), and AVG() expressions in a HASL specification for a given GSPN model.

COSMOS is a statistical model checker which takes as input a GSPN model and an HASL formula, and computes the estimated value of the performance indices specified as HASL formulas, together with the associated confidence intervals for the required confidence level and accuracy. A classical batch estimator is used to decide when the simulator has collected enough samples to reach the requested accuracy for the target HASL formulas. The language of GSPN accepted by COSMOS is extended to support transitions with both exponential distributions and general distribution. The latter are chosen from a predefined set of statistical distributions with positive support.

To speed up the simulation COSMOS transforms the GSPN model and the LHA into C++ code, and then builds and runs the compiled C++ application. Parallel simulation is supported, using multiple instances of the compiled application that run over multiple cores of the same machine, or multiple machines. In this case, the COSMOS application itself coordinates the parallel simulators. Recently COSMOS has been extended to support rare events handling with importance sampling [7, 6]. COSMOS is integrated into the CosyVerif [19] platform and uses its file format GrML both for GSPN and LHA. It also allows to use the graphical editor Coloane for drawing GSPN and LHA.

3. EXTENSION TO COLORED NETS

In the previous section we have presented HASL as a rich language to specify path-related performance indices of GSPN models, but HASL for GSPNs has also some limitations, for example the time needed for a token to complete a task described by a subnet (the so-called "passage time") is not easily specifiable. This is actually true also in more classical performance approaches: when the goal is the mean passage time computation we can apply Little's law, but when the goal is to compute the passage time distribution, things gets more complicated. It has been already recognized in the literature [2] that the distribution of the passage time requires the identification of the elements that circulate in a subnet. This is the main motivation for the extension of HASL, and of the associated tool COSMOS, to colored Petri nets presented in this section, since colored nets are nets in which each single token can have a unique identity. By extending HASL and COSMOS to work with colored nets we also gain the possibility of using HASL in some more realistic settings, where colored nets are typically used for their ability of representing complex models in a more compact and modular manner.

3.1 The colored formalism of Stochastic Symmetric Nets

The colored formalism considered for the extension of HASL is that of Stochastic Symmetric Nets (SSN)[1]. SSN are here introduced by means of the example in Fig. 2, representing a distributed database update protocol: a number of sites host processes that may update a shared distributed database, this must be done in mutual exclusion, and once an update has been performed by a process at one site it must notify the change to all the other sites, and wait until it receives an acknowledge message from each of them to then release the mutually exclusive access to the updated piece of information. The main difference of the SSN formalism with respect to GSPNs is the possibility to associate information to tokens (so that tokens are said to have *different colors*, instead of being all black and indistinguishable). Each place $p \in P$ has an associated color domain (a data type) denoted $cd(p)$ and each token in a given place has an associated value from $cd(p)$. Color domains are built from elementary types called *color classes* ($\{C_1, C_2, C_3, \ldots, C_n\}$), so that $cd(p) = C_1^{e_1} \times C_2^{e_2} \times \ldots \times C_n^{e_n}$, where e_j is the number

[1]SSN were previously known as *Stochastic Well-formed Nets*[9]; recently the untimed version of such formalism has been included in the High Level Petri Nets ISO standard **ISO/IEC 15909**, under the new name of Symmetric Nets (part 1, annex B.2).

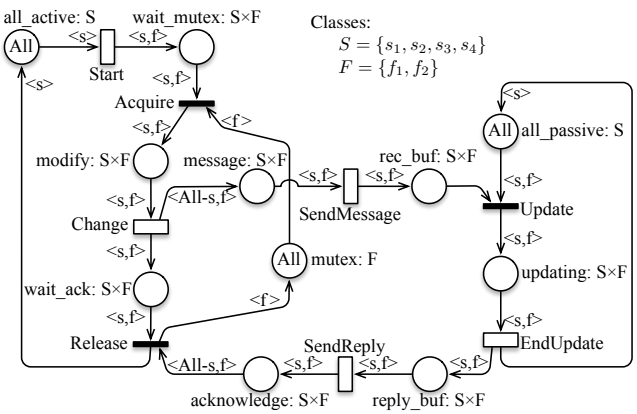


Figure 2: SSN model of a distributed database.

of times C_j appears in $cd(p)$. Color classes are finite, non empty and disjoint sets, they may be ordered (in this case a successor function is defined on the class, inducing a circular order among the elements in the class), and may be partitioned into (static) subclasses. In the database example of Fig. 2 there are two classes called S (for *site*) and F (for *file*), and the places have either color domain S, or F or $S \times F$. Also transitions have an associated color domain specified as a set of *typed variables* plus a *guard*, the variable types are color classes. A valid *transition binding* is an assignment of values to its variables, satisfying the predicate expressed by the *guard*. A pair *(transition,binding)* is called *transition instance*. All transitions in the example net have color domain $\{s : S, f : F\}$ and the guards are all set to *true* (which is the default and is omitted in the picture). Each arc connecting a place p and a transition t is labeled with an expression denoting a function $arcf : cd(t) \rightarrow Bag(cd(p))$ where $Bag(A)$ is the set of all possible multisets that may be built on set A. The valuation of $arcf$ given a legal binding of t gives the multiset of *colored tokens* to be withdrawn from (in case of input arc) or to be added to (in case of output arc) the place connected to that arc upon firing of such *transition instance*. The arc expressions in SSNs are built upon a limited set of primitive functions whose domains must be color classes. A typical arc expression takes the form of a linear combination of *function tuples* (denoted $\langle f_1, \ldots, f_n \rangle$), and each element of a tuple is either a *projection function*, denoted by a variable in the transition color domain (e.g. s and f appearing in the arc expression $\langle s, f \rangle$ labeling several arcs in the example net), a *successor function*, denoted $x++$ were x is a variable in the transition color domain whose type is an ordered class; a constant function returning all elements in a class (or subclass) denoted *classname.All*; a complement function denoted *classname.All*$-x$ where x is a variable of type *classname* in the transition color domain. Transition guards are boolean expressions whose terms take the form of *basic predicates*: $x = y$, $x \in subclass$, $d(x) = d(y)$ where x and y are variables of the transition with same type, and $d(x)$ denotes the static subclass x belongs to. The dynamic behavior of a SSN model is defined in terms of transition instances enabling and firing: the initial marking of the net in Fig. 2 enables $|S| \cdot |F|$ instances of transition *Start*, with binding $s = s_i, f = f_j, \forall s_i \in S, f_j \in F$ (the only condition for the enabling of such bindings is the presence of a token with color s_i in the input place *all_active*). Upon

firing of one enabled instance a token of color $\langle s_i \rangle$ is withdrawn from place *all_active* and a token of color $\langle s_i, f_j \rangle$ in place *wait_mutex*. The new reached marking enables only one instance of immediate transition *Acquire* with binding $s{=}s_i, f{=}f_j$: it is enabled because there is a token of color $\langle s_i, f_j \rangle$ in place *wait_mutex* and a token of color $\langle f_j \rangle$ in place *mutex* which are removed when the transition fires, while a token of color $\langle s_i, f_j \rangle$ is put into place *modify*. This enables the instance of transition *Change* with binding $\langle s_i, f_j \rangle$, which upon firing withdraws one token of color $\langle s_i, f_j \rangle$ from place *modify* and puts a similar token in place *wait_ack*, and $|S| - 1$ tokens $\langle s_k, f_j \rangle, s_k \in S, s_k \neq s_i$ in place *message*. A simulation trace of a SSN model consists of a sequence $m_i, ((t_i, b_i), \tau_i), m_{i+1}, ((t_{i+1}, b_{i+1}), \tau_{i+1}), \ldots$ of marking and transition instances (t_i, b_i) with the corresponding firing times τ_i.

3.2 Extending HASL to SSNs

The extension of HASL to support SSN models focuses on the definition of colored variables, of condition on colored markings, and on transition instances. Variables should be extended to be able to store colors as well, and edges should be able to discriminate between instances of the same transition with different bindings. The extension should be rich enough to allow also a reference to "coarser" quantities, for example, with reference to the place *modify* in the Database SSN, of color domain $S \times F$, we may want to refer to the total number of tokens, independently of the sites and files identities, or to the number of tokens with first color component equal to a specific site s_i.

To reach this goal each variable x of the LHA is indexed by a *color domain*, denoted as $cd(x)$. A *flow variable* x has a valuation $\bar{x}$ in $\mathbb{R}^{|cd(x)|}$. This implies that x is actually a (multidimensional) array of variables. Stable variables are further partitioned into *discrete* variables, whose valuation $\bar{x}$ is in $\mathbb{N}^{|cd(x)|}$ (and again this in general results in an array of variables), and *color* variables that represent just a color in a specified color class. Plain real and integer variables as in classical HASL are obtained by taking as colored domain the *neutral* one, which leads to *scalar* real (or integer) variables.

Transition names over edges should now be enriched to refer to transition instances, or to more coarser information about transition firings, although we should not forget that transition firing that synchronize with the automaton are the ones of the colored traces, that is to say, colored transition instances. To illustrate this aspect let us consider the three LHAs of Figure 3, in particular the self loops on l_0 that synchronize with the *Release* transition and increments variable x. In the leftmost LHA the transition is referred to simply as *Release*, and x is an integer variable, therefore x counts the number of occurrences of the transition *independently* from the binding. In the central LHA the transition is referred to as $Release\langle s, f\rangle[s, f = \langle s1, f1\rangle]$, which synchronizes only with the firing of *Release* for an instantiation of the SSN variables s and f equal to the colors *s1* and *f1* respectively, therefore variable x only counts the firings for that specific binding in the $S \times F$ domain of *Release*. Finally, the rightmost LHA considers colored transition instances, and counts the number of firings "per color", since now variable x is declared over the color domain $S \times F$, and the increment is for $x[s, f]$.

Expressions to be used in boolean conditions associated to locations may now refer to LHA colored variables and to

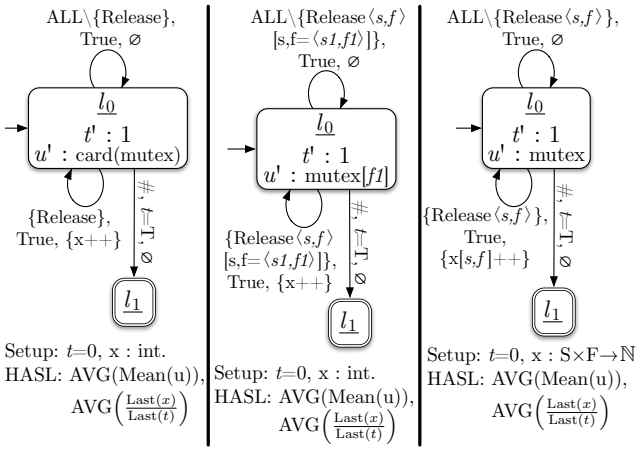


Figure 3: Three LHAs to compute the average number of tokens in place *mutex* and the throughput of *Release* at different granularity levels.

SSN markings, therefore they are typed with the color of the variable or of the SSN place. An expression typed with the neutral color is a scalar expression, while expressions in a color domain are vector expressions. Different types are incompatible, and cannot be mixed together. The language of *marking-dependent colored expressions* is then defined as:

$$exp ::= \alpha \mid p \mid x \mid exp\{+, -, *\}exp \mid (exp) \mid exp[color] \mid \text{card}(exp)$$

where $\alpha \in \mathbb{R}$ is a scalar constant, $p \in P$ is a place, x is a variable. Algebraic operators follow the usual scalar/vector semantics. The product between a scalar and a colored expression is allowed, and is interpreted as a scalar product with the same type of the colored operand. The *color* term is a colored token or a color variable. The *selection operator* $exp[color]$ gives the number of tokens of the specified color, as a scalar expression. The function card() gives the scalar cardinality of an expression (the total number of tokens, independently of their colors). When the color domain of a place or of a variable is the neutral domain, the selection operator $[color]$ may be omitted, since variables in the neutral domain are scalar variables.

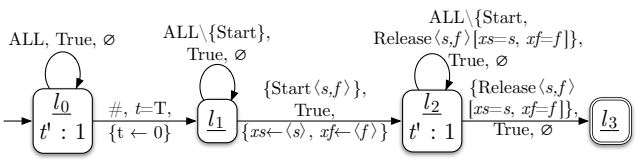
Boolean conditions are defined over the marking-dependent colored expressions, according to the following grammar:

$$lg ::= \text{true} \mid \neg lg \mid lg \wedge lg \mid exp \bowtie exp$$

where $\bowtie$ is a comparison operator. Comparisons on colored expressions are interpreted with the usual semantics of vectors comparison.

Color variables (stable variables that store a color) allows instead to store a color from a transition binding that synchronizes with an edge. Since color variables can be used to replace a color constant in an expression or in a transition specification, this allows to "match" transitions firing along a trace. To better illustrate this concept, let us consider the LHA of Figure 4, which has two color variables xs and xf of domain S and F respectively. The edge from l_1 to l_2 updates the color variables with the valuation of the variables s and f *of the SSN* as instantiated in the transition instance which synchronizes with the edge. Considering that location l_1 is entered when the time elapsed is equal to T, then

xs and xf store the identity of the first firing of transition *Start* after time T, the same identity is then used to select the "matched" firing of the *Release* transition.



Setup: t=0, colorvars xs : S and xf : F.
HASL expression: PROB().

Figure 4: An LHA to compute the probability that the first update starting after a period [0,T] ends before any other update can start.

To conclude the description of the colored extension of LHA we explain the properties specified by the four LHAs of Figure 3 and 4. The formulas for the leftmost LHA in Figure 3 compute, for the $[0, T]$ period, the throughput of transition *Release*, independently of the specific transition binding, as AVG(last(x)/last(t)), and the mean number of tokens in place mutex, independently of the color, with the formula AVG(mean(u)). Note that u is a flow variable of neutral color, that accumulates the cardinality of place *mutex* independently from the color of the tokens. The formulas for the central LHA compute two similar quantities, but, as explained before, the throughput is now accumulated for a single specific color, $s1, f1$, and the average number of tokens in place *mutex* only considers tokens of color $f1$. The rightmost LHA computes instead, for each formula, *a vector of performance indices*. Indeed the throughput is computed for each pair of s, f colors (combination of sites and files) and the number of tokens is computed "per file".

The formula PROB() in the LHA of Figure 4 computes the probability that the first update that starts after a period $[0, T]$ it will be able to terminate its update with a release of the mutex, before any other update starts. As explained before a color variable is used to match the firing of the considered *Start* transition with the corresponding firing of *Release*. Note the use of the flow variable t to compute the passage time distribution from the first *Start* after time T, until its *Release*, for those paths in which the matching *Release* happens before any other *Start*, this computation is correct because, thanks to colors, we are able to match exactly a *Start* instance with the corresponding instance of *Release*.

Color support in COSMOS.

Colors changes the way COSMOS manages markings and tokens. Colored places and bindings are supported by generating the proper C++ code that declares, for each domain d, its container data structure that represent a multiset. These multisets may have a co-domain as *double* (for flow variables) or *int* (for stable variables). The SSN support is heavily influenced by the design of the CosyVerif platform [19]. CosyVerif is a software environment made of several software tools, including the graphical interface *Coloane* for drawing SSN models and LHA automata, and COSMOS for the statistical model checking. This platform already have support for stochastic nets (which was the input format for COSMOS) and symmetric nets. COSMOS now takes as input an extension of the file format for the symmetric nets formalism of CosyVerif.

4. EXAMPLES

In this sections three examples are presented: they are taken from the literature and are used to illustrate the possibilities offered by the colored extension of COSMOS.

The hospital model.

In Fig. 5 we show our second SSN example modeling a hospital Emergency Department (ED) [3]. According to the description provided by [20], the ED patients are divided in three categories: patients requiring resuscitation (high priority), patients with major illnesses or injuries (medium priority) and patients with minor illnesses or injuries (low priority). This is modeled in the SSN model partitioning the color class P, representing the patients, in three static subclasses P_H, P_M and P_L; so that each static subclass identifies a different level of urgency of treatment (and hence of priority) of the corresponding patients.

All the healthy patients are in the place *Healthy*, while the sick patients heading towards the hospital are in place *Ill*. An healthy patient that falls sick is represented by the firing of (an instance of) transition *FallIll*, while the evaluation of sick patient reaching the hospital is modeled by the guards associated with transitions *HighPrio*, *MediumPrio* and *LowPrio*. All the high priority patients are immediately move into the resuscitation room (place *ResuscRoom*) to be stabilized: the stabilization process can start (transition *BtoStabilize*) iff there is at least one trauma team available (place *TraumaTeam* marked). When a patient is stabilized (transition *EtoStabilize*) then he/she is moved into the monitored room (place *MonitoredRoom*).

All the patients with medium priority, since they do not need for being stabilized, are immediately admitted (transition *MediumPrio*) into the monitored room where they are constantly monitored until the results of the blood and X-ray exams become available (transitions *EBloodExam* and *EX-Ray*); then the patient is treated or operated by a doctor according to the outcomes of his/her tests (transitions *ToDoctorM* and *ToSurgery*). A medical examination requires at least one doctor available (place *Doctors* marked), while a surgical operation at least one doctor and one operating room (place *OperatingRoom* marked) available.

Patients with low priority are moved in the waiting room (place *WaitingRoom*) and they are examined by a doctor only when no more higher priority patients need to be treated or operated (inhibitor arc from *ReadyT* to *ToDoctorL*).

Finally, transitions *DischargeL*, *DischargeM* and *DischargeRec* model the discharge of a patient from the ED.

LHA for computing passage time.

Now we describe how LHA extended with color can be used to compute passage time of selected paths.

The passage time distribution is an example of performance measure which is not straightforward to define on a PN model since it expresses the probability of the time required for a (specific) token to pass through a given subnet. However, it is a more informative measure than the average response time measures when we study the service level agreements or the safety-related requirements of a system.

Exploiting the expressiveness of LHA it is possible to de-

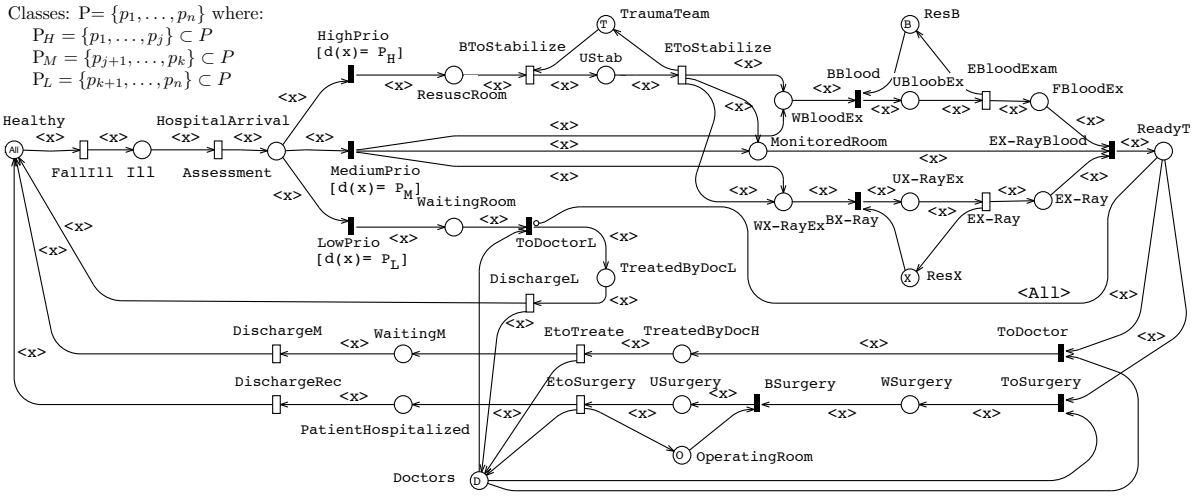


Figure 5: SSN model of patients flow in a hospital Emergency Department.

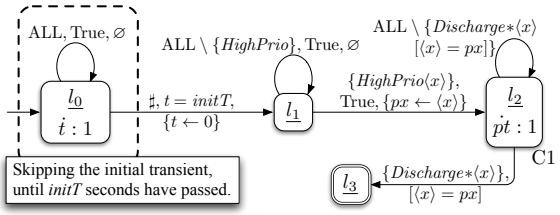
fine passage-time measures with quite complex constraints which may involve state conditions, transition firings, and performance characteristics of the model; it is also possible to decide in a very flexible way when the time count should be started and stopped on each path, including the possibility to interrupt the count in between the first start and final stop. This is much more powerful than classical state based specifications of passage time measures (which in their simplest form require to define the sets of entry, exit and forbidden states).

Hereafter, we describe two examples of passage-time measures:

H1: the CDF of the passage-time from arrival to discharge of a high priority patient given that at most n medium priority patients were in the hospital during his/her stay in the hospital;

H2: the CDF of the passage-time from arrival to discharge of a high priority patient given that the *average number* of medium priority patients has been less then n during his/her stay in the hospital.

The corresponding Colored LHAs are shown in Figs 6 and 7. In these two automata, the initial location l_0 is used only to skip an initial transient phase, so that a random trajectory of duration *initT* is simulated before starting the measure computation. After this initial phase the automata remains in l_1 until the first occurrence of the transition *HighPrio*, whose firing moves the automaton into the location l_2. On the edge connecting l_1 to l_2 the notation $px \leftarrow \langle x \rangle$ is used to "tag" a specific high priority patient assigning its specific color to the color variable x. Then, in the first automaton a path from l_2 is accepted by reaching final location l_3 as soon as the tagged patient is discharged from the hospital (i.e. transition *DischargeH* firing for the some colored assigned to px). Moreover, the condition $C1$ associated with l_2 assures that in all the accepted paths the number of medium priority patients in the hospital cannot be greater than n until the tagged patient is not discharged. Finally, the continuous flow variable pt is used to compute the time spent by the tagged patient inside the hospital.



Setup: $t = 0,\ pt = 0,\ px = \langle\rangle$
HASL expression: CDF(pt)
Condition C1: $card(\text{P.All} \setminus \{Healthy, Ill\}) \leq n$
Discharge∗ is the set $\{DischargeM, DischargeRec\}$.

Figure 6: Colored LHA for the H1 property.

Instead, in the second automaton a path from l_2 is accepted as soon as the tagged patient is discharged from the hospital provided that the average number of medium priority patients in the hospital has not been greater than n (i.e. edge notation $\frac{navg}{pt} \leq n$) during the patient stay.

In Fig.8 the CDF of passage time (with related confidence intervals) is plot for the properties H1 and H2 considering 20 patients (i.e. 4 high priority patients, 8 medium priority patients and 8 low priority patients). Note that the state space of this model becomes quite large increasing the number of patients (for instance, with 2 high priority patients, 3 medium priority patients and 6 low priority patients it has more than 1.5 billion states), so that the simulation approach becomes virtually the only applicable method. The time required to compute the data for Fig.8 with 20 patients is about 1.5 seconds on a laptop computer. The tool simulates approximatively 100,000 trajectories to ensure confidence intervals smaller than 0.001.

The Client-Server model.

The SSN model in Fig. 9(left) represents a client-server architecture where several similar clients whose identities are modeled by color class C, can require remote services to a server. When a client decides to send a request (transition *SendReq*), its message is queued in the server buffer to be pre-processed (place *pre_proc*); then it is served iff at

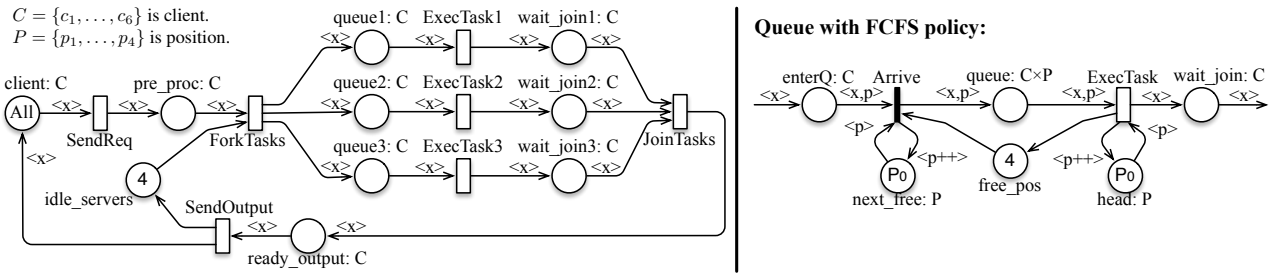


Figure 9: Client-server model.

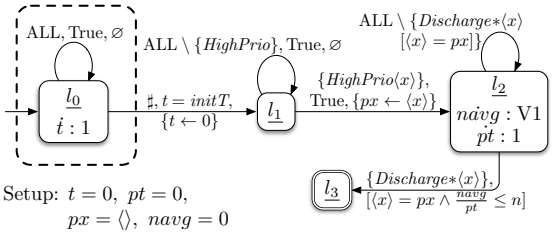


Figure 7: Colored LHA for the H2 property.

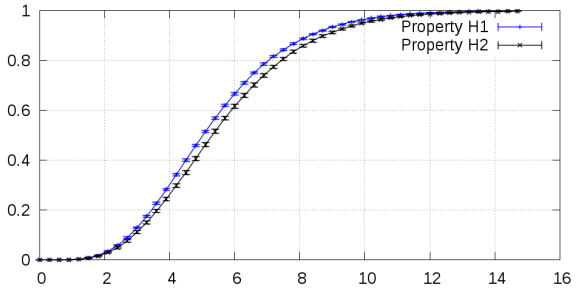


Figure 8: Passage time for the Hospital properties H1 and H2. For each point the related confidence interval is reported.

least one server is available (marking of place *idle_servers* greater than 0). Hence, three threads are generated (transition *ForkTask*) and executed in parallel (places $queue_i$ and transitions $ExecTask_i$). The computation result is sent back to the client (transiton *SendOutput*), only after all three threads end (transition *JoinTasks*).

The threads in places $queue_i$ are served using a Processor Sharing (PS) queueing policy (realized through a marking-dependent rate); while if we want to model a First-Come-First-Served (FCFS) queueing policy we have to replace the place $queue_i$ and its corresponding transitions $ExecTask_i$ with the submodel in Fig. 9 (right). In this submodel the place *queue*, with color domain $C \times P$, represents a circular array where the ordered color class P is used to encode the array indices. Places *head* and *next_free* encode the pointers to the first element in queue and to the first empty element, respectively.

The immediate transition *Arrive* assigns to a new thread in place *enterQ* the first empty position in the queue (i.e. marking of place *next_free*) if the queue limit is not reached (i.e. place *free_pos* is marked); and it updates the color of token in *next_Free* to the successor of p (i.e. denoted with the arc function $p++$). Instead, the firing of the transition *ExecTask* removes from place *queue* the token $\langle x, p\rangle$ with p equal to the color of the (unique) token in place *head*, then the color of the token in this place is updated to the successor of p.

LHA specification some performance indices.

The properties considered for this example are:

C1: throughput of *SendRequest* in transient and steady state;

C2: mean number of tokens in places *client* and *pre_proc* in transient and steady state;

C3: the CDF of the passage-time for the first client request processed by the server (from transition *ForkTask* to transition *JoinTask*) after the initial transient.

Observe that $C1$ and $C2$ are quite "standard" measures specified by means of reward function.

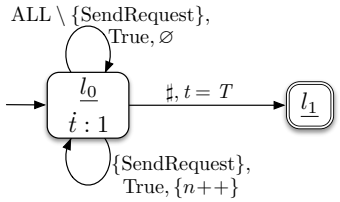
The Colored LHAs for these measures are shown in Figs 10, 11 and 12.

In details, the two automata for property $C1$ in Fig. 10 are different as regards to the dashed part used to skip an initial transient during the property computation. Then, the stable variable n is updated when transition *SendRequest* fires (independently by its colored binding). The computed formula is $AVG\frac{Last(n)}{Last(t)}$ where t is used to measure the time. For instance, considering 6 clients and 4 servers the throughput of the transition *SendRequest* assuming $initT = 100$ is 0.88629 ± 0.00019 when the threads in $queue_i$ are scheduled according to a PS policy, and 0.77278 ± 0.00017 when the threads in $queue_1$ are served using FCFS and the threads in $queue_2$ and $queue_3$ are served using PS.

In Fig. 11 the two automata show how to define the property $C2$ in transient and steady state. The flow variable p is used to store the number of tokens in places *client* and *pre_proc*. Hence, the computed formula is $AVG\frac{Last(p)}{Last(t)}$. For instance, the average number of tokens in these places with $initT$ equal to 100 time units, 6 clients and 4 servers is 0.89240 ± 0.00049 when the threads in $queue_i$ are served using PS, and 0.78369 ± 0.0003 when the threads in $queue_1$ are served using FCFS and the threads in $queue_2$ and $queue_3$ are served using PS.

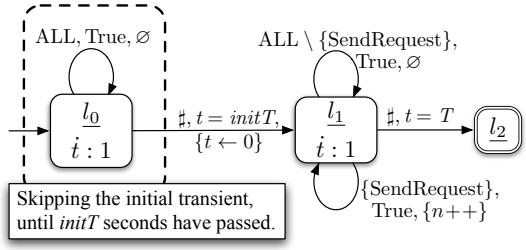
Finally, the last automaton in Fig.12 is an example of CDF of the passage-time in which a color variable (i.e. c)

1) *Throughput of* SendRequest *in transient* [0, T].



Setup: $t = 0$, $n = 0$.
HASL expression: $\text{AVG}\left(\frac{\text{Last}(n)}{\text{Last(t)}}\right)$

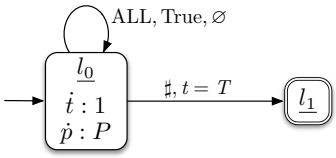
2) *Throughput of* SendRequest *in steady-state.*



Setup: $t = 0$, $n = 0$, $initT$ is the time skipped for the initial transient, T is the total sample time for each path.
HASL expression: $\text{AVG}\left(\frac{\text{Last}(n)}{\text{Last(t)}}\right)$

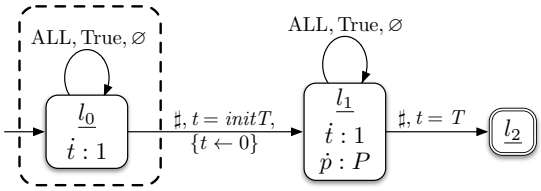
Figure 10: C1: throughput of SendRequest.

1) *Mean number of tokens in P in transient* [0, T].



Setup: $t = 0$, $p = 0$. The variable p accumulates in l_0 the token count of P.
P is the sum of tokens in *client* and *pre_proc*.
HASL expression: $\text{AVG}\left(\frac{\text{Last}(p)}{\text{Last(t)}}\right)$

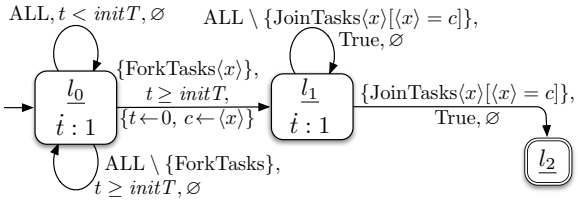
2) *Mean number of tokens in P in steady-state.*



Setup: $t = 0$, $p = 0$.
HASL expression: $\text{AVG}\left(\frac{\text{Last}(p)}{\text{Last(t)}}\right)$

Figure 11: C2: the mean number of tokens in P.

is used to tagged a generic client request (i.e. $c \leftarrow \langle x \rangle$), and shows another pattern for letting the initial transient to elapse. Indeed, the automaton moves from l_0 to l_1 only when the initial transient is elapsed ($t \geq initT$) and the transition *ForkTask* fires. The color associated with this



Setup: $t = 0$, $c = \langle\rangle$. c is a *color variable*.
HASL expression: CDF(t)

Figure 12: C3: passage time of the first observed client.

firing is stored in c, so that the corresponding client request is tagged. Then, the automaton reaches the final location l_2 iff the transition *JointTask* fires for the same tagged client request ($JointTask[\langle x \rangle = c]$).

In Fig. 13 the CDF of passage time computed by this automaton is shown considering: 1) PS queueing policy associated with places $queue_i$; 2)FCFS queueing policy associated with place $queue_1$ and PS to the others two places $queue_2$ and $queue_3$.

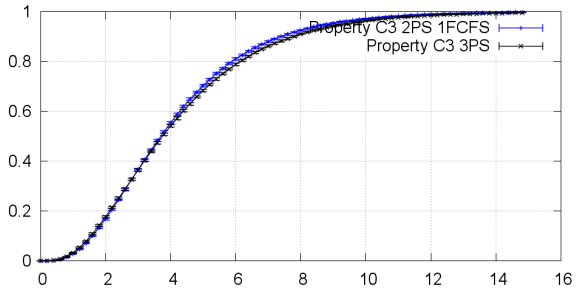


Figure 13: Passage time CDF of the first client request processed by the server after the initial transient for different task scheduling policies.

The Database model.

In this section the model in Fig. 2, already introduced in Section 3.1, is analyzed by means of COSMOS.

Two measures of interest are defined and computed:

D1: CDF of the time required for a change request to complete, under the condition that the time to acquire the mutex on the file to be changed does not exceed a given threshold and the time to send all the messages from the active site who modified the file and the passive sites is below a given threshold;

D2: the probability that a request for change, arriving when there are no other submitted requests, is the first to complete (i.e. no other request arriving later will overcome it)

The automata for D1 and D2 are shown in Figs. 14 and 16, respectively.

The automaton in Fig. 14 includes the usual pattern for letting the initial transient to elapse (from l_0 to l_1), after which the first occurrence of *Start* makes the automaton move to location l_2 while storing the binding in the two

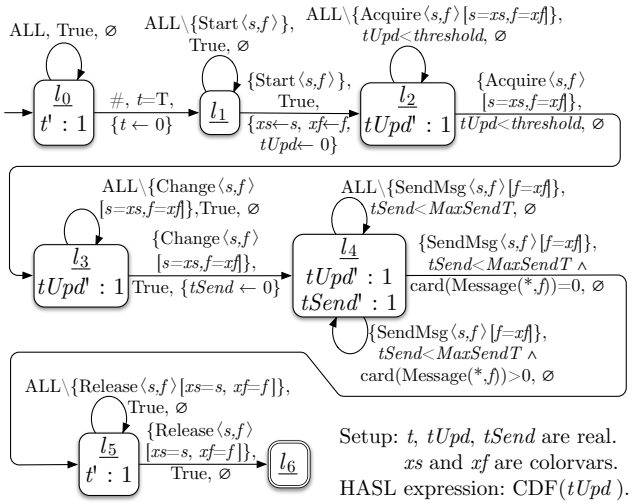


Figure 14: D1: passage time of a distributed update conditioned on a time thresholds on mutex acquisition and on update messages transmission.

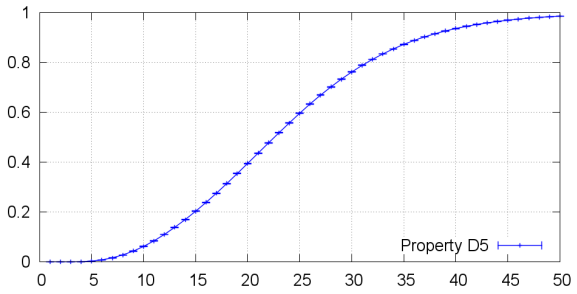
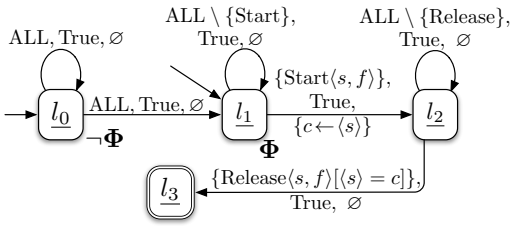


Figure 15: D1: CDF of the passage time of a distributed update conditioned on two time thresholds on intermediate steps.

color variables xs and xf. When entering l_2 the timer $tUpd$ is started and the control on the time elapsed until the firing of transition *Acquire* for the same binding is checked: if it exceeds the threshold the trace is discarded (condition $tUpd < threshold$ on all outgoing arcs). The firing of *Acquire* within the threshold brings the LHA to location l_3 where it remains until transition *Change* fires for the binding $\langle xs, xf\rangle$: this event starts the new timer $tSend$ and causes the LHA to move into location l_4. The purpose of location l_4 is to check that all messages from site xs to the other sites arrive to their destination within $MaxSendT$ time units (condition $tSend < MaxSendT$ on all outgoing arcs). The LHA moves on to location l_5 when an instance of transition *SendMsg* fires with variable f bound to xf which moves the last update message for file xf out of place *Message* (condition $card(Message(*, f)) = 0$) on the arc from l_4 to l_5. From l_5 the trace is accepted when transition *Release* fires with binding $\langle xs, xf\rangle$: timer $tUpd$ grows with rate 1 in all locations from l_2 to l_5 so that when the final location is reached it contains the time required to complete the update.

In Fig.15 the CDF computed for this property is reported.

The automaton in Fig. 16 has two initial locations to account for the possibility of different initial markings. Proposition Φ identifies a state where there are no ongoing updates (which is the case for the initial marking of the database example in Fig. 2). Hence the initial marking is matched with the appropriate initial location, l_0 or l_1; if it matches l_0 then the automaton remains in such state until a state is reached which satisfies Φ (that causes the automaton to move on to l_1), then at the first firing of transition *Start* the path recognition starts, recording the binding of s in the color variable c and moving on to location l_2. In l_2 the first occurrence of *Release* makes the automaton to either accept the path (if the binding of variable s is equal to c, moving to the final location l_3) or discard it (if s is bound to a value different from that stored in c). The measure is defined simply using the expression PROB(). This gives us the result of $0.5681588 \pm 9.02441 \cdot 10^{-4}$, computed with COSMOS using a confidence level of 99% on a total of 1.991.043 paths.



Setup: $t = 0$, OK $= 0$, $c = \langle\rangle$.
Condition $\mathbf{\Phi}$ is: all_active = All.
HASL expression: PROB()

Figure 16: D2: LHA for the probability of proper serialization in mutex acquisition.

5. CONCLUSIONS AND FUTURE WORK.

This paper presents an extension of the automata based language HASL, to express complex performance properties of SSN models in the COSMOS tool. The use of an High Level (Stochastic) Petri Net formalism allows to describe realistic systems, on the other hand the possibility to use LHA extended with color-indexed variables and propositions involving colored markings provides a rich and powerful language for property specification, increasing the practical applicability of the COSMOS simulator. This has been demonstrated on three examples (taken from the literature).

The current implementation of the COSMOS extension to SSNs and HASL does not embed a number of optimizations that could be implemented taking advantage of the specific symmetric properties of the SSN models. The type of arc functions that characterize the formalism naturally leads to several possibilities for the optimization of the future event list updating and handling. The approach proposed in [13] and implemented in the GreatSPN software tool can be adapted and improved in COSMOS: this includes methods to efficiently determine which transition instances are newly enabled or disabled, based on a preliminary analysis of the net structure; the techniques must be adapted to take into account the COSMOS architecture, which is based on the generation of model specific code to be compiled with the simulator engine core.

6. REFERENCES

[1] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model-checking continuous-time Markov chains. *ACM Trans. Comput. Logic*, 1(1):162–170, 2000.

[2] G. Balbo, M. Beccuti, M. De Pierro, and G. Franceschinis. First Passage Time Computation in Tagged GSPNs with Queue Places. *The Computer Journal*, 2010. First published online July 22, 2010.

[3] G. Balbo, M. Beccuti, M. D. Pierro, and G. Franceschinis. Computing first passage time distributions in stochastic well-formed nets. In *ICPE'11 - Second Joint WOSP/SIPEW International Conference on Performance Engineering*, pages 7–18, Karlsruhe, Germany, March 2011.

[4] P. Ballarini, H. Djafri, M. Duflot, S. Haddad, and N. Pekergin. COSMOS: a statistical model checker for the hybrid automata stochastic logic. In *Proceedings of the 8th International Conference on Quantitative Evaluation of Systems (QEST'11)*, pages 143–144, Aachen, Germany, Sept. 2011. IEEE Computer Society Press.

[5] P. Ballarini, H. Djafri, M. Duflot, S. Haddad, and N. Pekergin. HASL: An expressive language for statistical verification of stochastic models. In *Proceedings of the 5th International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS'11)*, pages 306–315, Cachan, France, May 2011.

[6] B. Barbot, S. Haddad, and C. Picaronny. Coupling and importance sampling for statistical model checking. In C. Flanagan and B. König, editors, *TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2012.

[7] B. Barbot, S. Haddad, and C. Picaronny. Importance sampling for model checking of continuous time markov chains. In P. Dini and P. Lorenz, editors, *SIMUL 2012*, pages 30–35, Lisbon, Portugal, Novembre 2012. IARIA.

[8] T. Chen, T. Han, J.-P. Katoen, and A. Mereacre. Quantitative Model Checking of Continuous-Time Markov Chains Against Timed Automata Specifications. *Logic in Computer Science, Symposium on*, 0:309–318, 2009.

[9] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic well-formed coloured nets for symmetric modelling applications. *IEEE Trans. on Computers*, 42(11):1343–1360, nov 1993.

[10] A. Clark and S. Gilmore. State-aware performance analysis with extended stochastic probes. In *Proceedings of the 5th European Performance Engineering Workshop on Computer Performance Engineering*, EPEW '08, pages 125–140, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] S. Donatelli, S. Haddad, and J. Sproston. Model checking timed and stochastic properties with $\mathrm{CSL}^{\mathrm{TA}}$. *IEEE Trans. Softw. Eng.*, 35(2):224–240, 2009.

[12] E. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241 – 266, 1982.

[13] R. Gaeta. Efficient discrete-event simulation of colored petri nets. *IEEE Trans. Softw. Eng.*, 22(9):629–639, Sept. 1996.

[14] J. Hillston. Process algebras for quantitative analysis. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, pages 239–248, Washington, DC, USA, 2005. IEEE Computer Society.

[15] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modeling with Generalized Stochastic Petri Nets*. J. Wiley, New York, NY, USA, 1995.

[16] W. D. Obal, II and W. H. Sanders. State-space support for path-based reward variables. *Perform. Eval.*, 35:233–251, May 1999.

[17] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

[18] K. Sen, M. Viswanathan, and G. Agha. Statistical model checking of black-box probabilistic systems. In *In 16th conference on Computer Aided Verification (CAVÕ04), volume 3114 of LNCS*, pages 202–215. Springer, 2004.

[19] The CosyVerif Group. Cosyverif. `http://www.cosyverif.org`.

[20] S. Wau Men Au-Yeung. *Response Times in Healthcare Systems*. PhD thesis, Imperial College, London, 2008. `pubs.doc.ic.ac.uk/response-times-in-healthcare`.

[21] H. L. S. Younes. Ymer: A statistical model checker. In *COMPUTER AIDED VERIFICATION. LNCS*, pages 429–433. Springer, 2005.