Learning generalized semi-Markov processes: From stochastic discrete event systems to testing and verification

André de Matos Pedro, Maria João Frade and Simão Melo de Sousa

Technical Report Series: DCC-2012-01
Version 1.1

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Laboratório de Inteligência Artificial e Ciência de Computadores

Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre, 1021/1055,
4169-007 PORTO,
PORTUGAL

Tel: 220 402 900 Fax: 220 402 950
http://www.dcc.fc.up.pt/Pubs/

Learning generalized semi-Markov processes: From stochastic discrete event systems to testing and verification

André de Matos Pedro[1], Maria João Frade[1] and Simão Melo de Sousa[3⋆]

[1] University of Minho, Braga, Portugal,
pg15753@alunos.uminho.pt, mjf@di.uminho.pt,
[2] LIACC, University of Beira Interior, Covilhã, Portugal,
desousa@ubi.pt

Abstract. Discrete event systems (DES) are widely used to model a set of practical systems, such as: industrial systems, computer systems, and also traffic systems. This report explores an extension of discrete event systems with an emphasis on stochastic processes, commonly called stochastic discrete event systems (SDES). There is a need to establish a stochastic abstraction and a model for SDES through a generalized semi-Markov processes (GSMPs) and respectively a stochastic timed automaton. In this report we propose a novel algorithm to learn GSMPs from sample executions that can be used for quantitative analysis and verification in the context of model checking. We demonstrate that the proposed learning algorithm can correctly identify the GSMPs given sufficient samples. This report also presents a Matlab toolbox for our algorithm and a case study of the stochastic analysis for a multiprocessor system scheduler.

1 Introduction

Stochastic processes are commonly used as an approach to describe and make a quantitative evaluation of more abstract models which may be described by a high-level specification. *Stochastic discrete event systems* (SDES) are models with a wide practical application [2], where the evolution of states are guided by events that are triggered by a stochastic process. The canonical example is a simple queuing system with a single server where the enqueue and the dequeue events are controlled by a stochastic process [17]. For instance, consider a post office with only one employer, and therefore one queue for letters, packages, etc. The customers arrive at the post office, deposit the goods in the queue, are attended by the postal worker and then leave the post office. We can think of the arrival and departure of a customer as two separate events with associated ratings, without synchronization between these events. This is clearly an example of SDES, where coping with asynchronous events and stochastic processes are their major advantage. When a model is evaluated we can use it for the design phase and subsequently make an implementation. However, even if a model is validated this does not imply that the implementation is in conformity with the model. This is normally due to bugs in the implementation, wrong interpretation of the model, or possibly, wrong approximations in the construction of the stochastic model. Unfortunately techniques like testing for discovering these errors are unlikely to be sufficient due to the difficulty of achieving a complete or total coverage.

This report is concerned with how these models can be derived from sample executions provided by an implementation in order to verify them. There are several learning algorithms for learning stochastic languages [1,11,18], including a learning algorithm for continuous-time Markov processes [16], but there is no algorithm in the case of processes that do not hold the Markov property such as *generalized semi-Markov processes* (GSMPs). Although the continuous-time Markov processes could in some way be adapted for these cases, the coverage of SDES would in fact be small. The use of GSMPs may cover a wider set of problems, but they could become more complex and quasi analytically intractable. However, establishing the *generalized semi-Markov process* (GSMP) as the

⋆ This work was supported in part by the FCT CANTE project (Ref[a]PTPC/EIA-CCO/101904/2008).

abstraction process to model the behavior of SDES for learning is statistically an ambitious goal due to its expressiveness.

In this report, we address the problem of learning GSMPs that are the most known extensive stochastic processes when lifetimes can be governed by any continuous probabilistic distributions [5]. From classical Markov processes, the exponential distributions are not enough to model the lifetime of a product (e.g., a electronic component life) [14] or even a computer process [10]. The learning algorithm we shall present infers a GSMP from a given set of trajectories and therefore must be capable of inferring the model by running the deployed system in a test phase and of learning trajectories according to the observed distributions. The learned stochastic automaton that is generated by a GSMP is a model that can be used by existing statistical model-checkers [13,21,20,3] and by the existing performance evaluation tools for further analysis and thereby ultimately helping to find bugs in the post-implementation phase. Learning algorithm for GSMPs may also potentially be used to perform automatic verification for stochastic systems.

In addition we also establish the correctness of our algorithm. We ensure that, in the limit, when the samples grow infinitely the learned model converges to the original model. Thus, a set of conditions like the definition of inclusion of a prefix tree in a GSMP have to be ensured as well as the definition of probability measure of paths.

This report is organized as follows. In section 2 some preliminary definitions are given in order to establish the learning algorithm in section 3. In section 4 we demonstrate the correction of our algorithm. In section 5, the tool and a practical case study are presented. In the final section 6 we give our conclusions and discuss directions for further work.

2 Preliminaries

SDES are *discrete event systems* with a casual relationships between events and states. Their dynamic behavior can be described by a stochastic process, more precisely a Markov or a semi-Markov process. We shall present the GSMP as an abstraction of SDES. GSMP allow a wide coverage of treatable stochastic processes and therefore SDES. The semi-Markov property of GSMP is due to the fact that at the state transition instant, the next state is determined not only by the current state but also by the event that just occurred.

Definition 1. *A* generalized semi-Markov process *is a stochastic process* $\{X(t)\}$ *with state space* X*, generated by a* stochastic timed automaton $(\mathcal{X}, \mathcal{E}, \Gamma, p, p_0, G)$*, where*

$\mathcal{X}$	*is a finite state space,*
$\mathcal{E}$	*is a finite event set,*
$\Gamma(x)$	*is a set of feasible or enabled events, defined for every* $x \in \mathcal{X}$*, with* $\Gamma(x) \subseteq \mathcal{E}$*,*
$p(x'; x, e')$	*is a state transition probability, defined for every* $x, x' \in \mathcal{X}$ *and* $e' \in \mathcal{E}$*, and such that* $p(x'; x, e') = 0,\ \forall e' \notin \Gamma(x)$*,*
$p_0(x)$	*is the* pmf *(probability mass function)* $Pr[X_0 = x],\ x \in \mathcal{X}$*, of the initial state* X_0*, and*
$G = \{G_i : i \in \mathcal{E}\}$	*is a stochastic clock structure where* G_i *is a* cdf *(cumulative distribution function) for each event* $i \in \mathcal{E}$*.*

The stochastic clock structure has k *random variables associated for each event* i *with distribution function* G_i *and is denoted by* $V_{i,k} \sim G_i$ *(i.e., the random sequences* $\{V_{i,k}\}$ *generated through* G_i*), where* $k = 1, 2, ...,\ i \in \mathcal{E}$*, and the tilde* ($\sim$) *notation denotes "with distribution". More precisely* G_i *is defined by* cdf $F_{V_{i,k}}(t) = Pr[V_{i,k} \leq t]$.

We begin with the notation of random variables in order to describe how SDES evolves. The random variables are as follows: X is the current state; E is the most recent event (causing the

transition into state X); T is the most recent event time (the time of the event E); N_i is the current score of event i; and Y_i is the current clock value of event i. We also use the (') notation to denote the next state X'; triggering event E'; next event time T'; next score of i, N_i'; and next clock of i, Y_i'.

Given a stochastic process with a state sequence $\{X_0, X_1, X_2, ...\}$ generated by a STA there is a transition mechanism that depends on two observations, $X = x$ and $E' = e'$ (note that X, E' are random variables and x, e' the produced samples). Thus, a GSMP is governed by the transition function $p(x'; x, e')$ that represents the probability that given a state x and an event e' there is a transition to state x'. However e' is driven by a stochastic event sequence $\{E_1, E_2, E_3, ...\}$ that is generated by

$$E' = arg \min_{i \in \Gamma(X)} \{Y_i\} \tag{1}$$

with the stochastic clock values Y_i, $i \in \mathcal{E}$, defined by

$$Y_i' = \begin{cases} Y_i - Y^* \text{ if } i \neq E' \text{ and } i \in \Gamma(X) \\ V_{i,N_i+1} \quad \text{if } i = E' \text{ or } i \notin \Gamma(X) \end{cases} \quad i \in \Gamma(X') \tag{2}$$

where the interevent time Y^* is defined as

$$Y^* = \min_{i \in \Gamma(X)} \{Y_i\} \tag{3}$$

the event scores N_i, $i \in \mathcal{E}$, are defined through

$$N_i' = \begin{cases} N_i + 1 \text{ if } i = E' \text{ or } i \neq \Gamma(X') \\ N_i \qquad\qquad \text{otherwise} \end{cases} \quad i \in \Gamma(X') \tag{4}$$

and

$$V_{i,k} \sim G_i \tag{5}$$

where $V_{i,k}$ is a random variable with distribution G_i. The initial conditions of the STA are defined as follows:

. $X_0 \sim p_0(x)$,
. $Y_i = V_{i,1}$, $N_i = 1$ if $i \in \Gamma(X_0)$, and
. Y_i is undefined and $N_i = 0$ if $i \notin \Gamma(X_0)$.

To avoid having two events simultaneously trigger a transition, we focus our attention on equation 1 that by formulation describes that only one event can be triggered at each time. Our definition of E' thus yields a unique triggering event, and event times are updated through $T' = T + Y^*$.

2.1 Trajectories and probability measure

An execution of a stochastic discrete event system produces samples as output values which we denote by a *sample execution*. In other words a sample execution is a *path* obtained from an execution, i.e., the evolution of SDES over time is captured by a path. Typically, the executions of SDES are infinite, therefore in order to achieve finite trajectories a methodological approach must be devised (as we see later). An *infinite path* of a GSMP $(\mathcal{X}, \mathcal{E}, \Gamma, p, p_0, G)$ is composed of a set of segments that we call *piecewise constant* and can therefore be represented as a sequence $\sigma = \{s_0, \langle e_1, t_1 \rangle, s_1, \langle e_2, t_2 \rangle, s_2, \langle e_3, t_3 \rangle, \cdots\}$ where $s_i \in \mathcal{X}$ is a state, $e_i \in \mathcal{E}$ is an event occurring at time $\sum_{x=1}^{i} t_x = \tau$, and $t_i \in \mathbb{R}_{\geq 0}$ is the holding time of each event, for all $i \in \mathbb{N}$. The initial state of the sequence is $s = s_0$. A *finite path* of length n of a GSMP is defined by the sequence $\sigma = \{s_0, \langle e_1, t_1 \rangle, s_1, \langle e_2, t_2 \rangle, s_2, \langle e_3, t_3 \rangle, \cdots, s_n, \langle \cdot, \infty \rangle\}$, where s_n is an absorbing state, meaning that no events can occur in s_n. An infinite path can be convergent, however, an infinite sequence of events would have to occur in a finite amount of time, which is unrealistic for any physical system.

The probability measure from a set of paths need the establishment of a prefix $\sigma_{\leq\tau} = \{s_0, \langle e_1, t_1\rangle, s_1, \langle e_2, t_2\rangle, ..., s_k, \langle e_{k+1}, t_{k+1}\rangle\}$ based on the infinite sequence $\sigma = \{s_0, \langle e_1, t_1\rangle, s_1, \langle e_2, t_2\rangle, \cdots\}$. A set of paths with prefix p is denoted by $Path(p)$, where p shall be $\sigma_{\leq\tau}$. Therefore, the sets of trajectories must be measurable. From the definition (1) a GSMP has associated with each event $e \in \mathcal{E}$ a positive trigger time distribution G_e, and a state transition probability matrix $p(x'; x, e)$. The *pdf* for G_e, $h_e(\cdot; \sigma_{\leq\tau})$, can depend on the entire execution history that is characterized by $\sigma_{\leq\tau}$. Thus, the holding time distribution, for each event $e \in \mathcal{E}$, defined as $H_e(t; \sigma_{\leq\tau})$ is subdivided in two clock types, termed the *new clock* and the *old clock*. The first is characterized by the probability that e triggers within t time units (i.e., a new clock) which is denoted by $Pr[Y_e < t \mid \sigma_{\leq\tau}]$. The second characterizes the probability that event e triggering in the next t time units when e has already been enabled for u_e time units (i.e., an old clock) which is denoted by $Pr[Y_e < u_e + t \mid Y_e > u_e, \sigma_{\leq\tau}]$. A sample execution of a GSMP can be measured using the holding time distribution,

$$H_e(t; u_e, \sigma_{\leq\tau}) = \begin{cases} \int_0^t h_e(x; \sigma_{\leq\tau})\, dx & \text{if } u_e = 0 \\ 1 - \frac{1-H_e\left(u_e+t;\, 0, \sigma_{\leq\tau}\right)}{1-H_e\left(u_e;\, 0, \sigma_{\leq\tau}\right)} & \text{if } u_e > 0 \end{cases} \tag{6}$$

and a next-state probability matrix $P_e(s'; \sigma_{\leq\tau})$ that is equal to $p(s'; s, e')$ due to time homogeneity of GSMPs under consideration [8]. The probability measure μ for a *cylinder set* $C(\sigma_{\leq\tau}, \langle E_k, Y_k^*\rangle, X_k, ..., X_{n-1}, \langle E_n, Y_n^*\rangle, X_n)$, accordingly to [21], can be defined recursively as

$$\begin{aligned} &\mu(C(\sigma_{\leq\tau}, \langle E_k, Y_k^*\rangle, X_k, ..., X_{n-1}, \langle E_n, Y_n^*\rangle, X_n)) = \\ &\quad P_e(s'; \sigma_{\leq\tau}) \cdot H_e(t; \cdot, \sigma_{\leq\tau}) \cdot \mu(C(\sigma_{\leq\tau} \oplus (\langle e, t\rangle, s'), \\ &\qquad \left\langle E_{k+1}, Y_{k+1}^*\right\rangle, X_{k+1}, ..., X_{n-1}, \langle E_n, Y_n^*\rangle, X_n)) \end{aligned} \tag{7}$$

where the recursive base case is $\mu(C(s_0, \langle E_1, Y_1^*\rangle, X_1, ..., X_{n-1}, \langle E_n, Y_n^*\rangle, X_n)) = 1$. The enabled events in a state race to be the first to trigger, the event that triggers causes a transition to a state $s' \in \mathcal{X}$ according to the next-state probability matrix for the triggering event. The GSMP is considered as analytically intractable and the probability measure formulation is not at all intuitive due to the execution history.

2.2 Exemplification of simulation procedure

The following example is now given in order to illustrate the concept of simulation of a GSMP in terms of the scheduling of events and the competition between events. Let $M = (\mathcal{X}, \mathcal{E}, \Gamma, p, p_0, G)$ be a model of a $M/M/1/n$ queue (Kendall notation), where

- $\mathcal{X} = \{0, 1, 2, 3, ..., n\}$,
- $\mathcal{E} = \{\lambda, \mu\}$,
- $\Gamma(0) = \{\lambda\}$, $\Gamma(1) = \{\lambda, \mu\}$, ..., $\Gamma(n-1) = \{\lambda, \mu\}$, $\Gamma(n) = \{\mu\}$,
- $p_\lambda(s'; s) = 1$ if $s' = i$ and $s' = i + 1$ otherwise, $p_\lambda(s'; s) = 0$
- $p_\mu(s'; s) = 1$ if $s' = i$ and $s' = i - 1$ otherwise, $p_\mu(s'; s) = 0$, with $0 \leq i \leq n$,
- $p_0(0) = 1$ otherwise, $p_0(j) = 0$, for all $0 < j \leq n$, and
- $G_\lambda(x) = 1 - e^{-0.1x}$, $G_\mu(x) = 1 - e^{-\left(\frac{x}{0.5}\right)^{0.1}}$.

The simulation of SDES is informally described with the simple four steps as follows:

1. Generate clock samples for feasible events i according to clock structure G_i, $V_{i,k}$.
2. Compare the current clock values for feasible events i and determine the triggering event $E' = e^*$, i.e., the event with minor lifetime.
3. Update the process for next state s' according to the triggered event e^*, p(s'; s, e*).
4. Update the clock values Y_i and scores N_i respectively.

Thus, the presented example has the following possible executions. First, the system can only start in state 0 accordingly to p_0. There is a single event λ (accordingly with $\Gamma(0)$) that denoted the *enqueue*. A sample clock would be generated accordingly with G_λ. Supposing that the value 0.76 is generated then the execution path begins with $\sigma_{\leq 0.76} = \{\langle 1, \lambda, 0.76\rangle\}$. Next, there are two feasible events, λ enqueue and μ *dequeue*. These events compete with each other to trigger the event with minor lifetime that in this case is λ with new value 0.45 versus μ with new value 0.68. The current execution path is $\sigma_{\leq 0.76+0.45} = \{\langle 1, \lambda, 0.76\rangle\ \langle 2, \lambda, 0.45\rangle\}$. Then, assuming that the event μ is triggered with value $0.68 - 0.45 = 0.23$, λ shall have a new value 0.41. Lastly, after these three execution traces the path shall be $\sigma_{\leq 1.44} = \{\langle 1, \lambda, 0.76\rangle\ \langle 2, \lambda, 0.45\rangle, \langle 1, \mu, 0.23\rangle\}$.[3] We can, however, calculate the probability of path $\sigma_{\leq 1.44}$ which can be measured as $\mu(C(\sigma_{\leq 1.44}, \langle E_k, Y_k^*\rangle, X_k, ..., X_{n-1}, \langle E_n, Y_n^*\rangle, X_n))$.

3 Learning stochastic automaton

The learning of stochastic automata covered in this report falls in the category of stochastic language identification [1,11,18,16]. For most of the methods in this category, the identified stochastic languages have some particularities: the inference of a stochastic language is normally based on sample executions, i.e., these samples are a particular multi-set of the original language to identify, and the inference has as target the identification of the language in the limit, i.e., if the number of samples tends towards infinity then the learned language will converge to the same as the language that generated the sample [9]. Thus, the learning of stochastic languages essentially follows a common method. First, establishing an equivalent relation between the states, then constructing a prefix tree from samples provided by the original stochastic language, and lastly describing an algorithm for the merge of equivalent states which is called *state merge*.

We shall now present an algorithm for learning GSMPs, which is a particular case of stochastic automata (addressed in the master thesis [4]). We begin with the formulation of *prefix tree stochastic automata* (PSA) in order to establish the inclusion of a prefix tree in a stochastic automaton. We next present some definitions that will be used in the algorithm. After this, we describe in detail the learning algorithm.

3.1 Inclusion of prefix tree in stochastic automaton and the state equivalency

In order to establish the correction of the learning algorithm, a relation between the language accepted by the prefix tree and the event language accepted by the GSMP must be defined. Thus, we need to establish that a set of *finite paths* acquired by a GSMP is equivalent to a set of finite paths accepted by the prefix tree. Based on the definition (2) of the *prefix tree stochastic automata* it is shown by definition that a prefix tree constructed from a set of finite paths is a base case of a GSMP. However, only the relation between the data structures is ensured with this definition. Therefore, we also need to establish a correction of the state merge algorithm (as shall be shown in the next section).

We introduce some notations in order to describe and establish the definitions (prefix tree, trajectories, prefix tree stochastic automata, and the stable equivalence relation) and the structure of the algorithms. The definitions are based on symbol ('$\|$') that represents a path with respect to an element of a particular set $(\mathcal{X}; \mathcal{E}; G)$ and brackets ('['';']') a sequential identification, as follows:

- $\sigma\|_{\mathcal{X}}[s, i]$ is the i^{th} state of the state sequence that begins in state s,
- $\sigma\|_{\mathcal{E}}[s, i]$ is the i^{th} event of the event sequence that begins in state s,
- $\sigma\|_{G}[s, i]$ is the i^{th} holding time of the event sequence $(\sigma\|_{\mathcal{E}}[s, i])$ that begin in s state,
- $\eta(\sigma\|_{\mathcal{E}}[s, i]) = \sigma\|_{\mathcal{X}}[s, i-1]$ is a function that returns the state associated to an event e_i,

[3] The execution of those systems is typically infinite, therefore, given a finite execution time we cannot make a infinite event sequence, which turns impossible to have finite executions in physical systems.

. $\varepsilon(\sigma\|_{\mathcal{X}}[s,i]) = \sigma\|_{\mathcal{E}}[s,i+1]$ is a function that given a state of a path returns its associated event, and
. $\delta(\sigma\|_{\mathcal{E}}[s,i]) = \sigma\|_{G}[s,i]$ is a function that given an event $\sigma\|_{\mathcal{E}}[s,i]$ returns its holding time $\sigma\|_{G}[s,i]$.

A sequence of events $\langle e_1, e_2, e_3, \ldots, e_k\rangle$ produced by the prefix tree that accepts the prefix $\sigma_{\leq\tau} = \{s_0, \langle e_1, t_1\rangle, s_1, \langle e_2, t_2\rangle, ..., \langle e_k, t_k\rangle, s_k\}$ is denoted by $\sigma_{\leq\tau}\|_{\mathcal{E}}$. A *prefix tree* that has an acceptor $Path(\sigma_{\leq\tau})$ (a set of paths with prefix $\sigma_{\leq\tau}$) is a tree $Pr(Path(\sigma_{\leq\tau})) = (\mathcal{F}, \mathcal{Q}, \rho, \varrho, \delta)$, where $\mathcal{F}$ is a set of leaf nodes of the prefix tree (i.e., $\mathcal{F} = Path(\sigma_{\leq\tau}\|_{\mathcal{E}})$), $\mathcal{Q}$ is the set of nodes of the prefix tree composed by the sequence of events from $Path(\sigma_{\leq\tau}\|_{\mathcal{E}})$ (i.e., $\mathcal{Q}$ represents all accepted sequences in the prefix tree), $\rho : \mathcal{Q} \to [0,1]$ is the function that associate the expectation value for each node $n \in \mathcal{Q}$, $\varrho : \mathcal{Q} \to \mathbb{R}_{>0} \times ... \times \mathbb{R}_{>0}$ is the function that associate each node with a n-tuple of clock values, and $\delta : \mathcal{Q} \to \mathcal{Q} \cup \perp$ is the transition function which have the following definition,

$$\begin{aligned}
\delta(s,\lambda) &= s \text{ where } \lambda \text{ is the empty string,}\\
\delta(s,e) &= \perp \text{ if } \delta(s,e) \text{ is not defined, and}\\
\delta(s,xe) &= \delta(\delta(s,x),e), \text{ where } x \in Q \text{ and } e \in \mathcal{E},\\
\delta(s,xe) &= \perp \text{ if } \delta(s,x) = \perp \text{ or } \delta(\delta(s,x),e) \text{ is undefined.}
\end{aligned}$$

We describe some definitions to simplify the notations of the relation between paths and the prefix tree, as follows:

. $\tau(s, xe_i)$ is a function that gives the set of feasible or enabled events $\{s, x \in \mathcal{Q}, e_i \in \mathcal{E} \mid \forall y : \delta(\delta(s, x\,e_i), y) \neq \perp\}$ of a given event sequence $x\,e_i$ from a prefix tree $Pr(Path(\sigma_{\leq\tau}))$, for instance from a set of sequences $\{x\,e_1\,e_2, x\,e_1\,e_3, ...\}$ we get $\{e_2, e_3, ...\}$,
. a map function $\nu(\sigma\|_{\mathcal{X}}[s,i]) = u$, where $u \in \mathcal{Q}$ is a sequence of events accepted by the prefix tree $Pr(Path(\sigma_{\leq\tau}))$, and
. $\varrho(s, xe_i)$ is a function that gives the holding times associated at each word xe_i in a prefix tree $Pr(Path(\sigma_{\leq\tau}))$.

Now, we present the definition (2), an inclusion of a prefix tree in a GSMP. One says that a prefix tree $Pr(Path(\sigma_{\leq\tau}))$ is a particular case of a GSMP, or in other words a stochastic automaton. However, we also need to ensure by definition that after the state merge, the original model remains a GSMP (as we will see later).

Definition 2. *The prefix tree $Pr(Path(\sigma_{\leq\tau})) = (\mathcal{F}, \mathcal{Q}, \rho, \varrho, \delta)$ for a set of multiple paths $Path(\sigma_{\leq\tau})$ is a particular stochastic automaton, i.e., $PSA(Path(\sigma_{\leq\tau})) = (\mathcal{X}, \mathcal{E}, \Gamma, p, p_0, G)$, where*

. $\mathcal{X} = \mathcal{Q}$,
. $\mathcal{E}$ *is the set of single and unique events in the* $\mathcal{F}$ *set,*
. $\Gamma(s_i) = \tau(s, \nu(s_i))$,
. $p(s'; s, e^*) = \begin{cases} 1 & \text{if } \delta(\nu(s), e^*) \neq \perp \text{ and } \nu(s') \neq \perp \\ 0 & \text{otherwise} \end{cases}$,
. $p_0(s) = 1$, *and*
. G *is a set of distributions estimated by sample clocks associated on each event, given by function* ϱ.

The $PSA(Path(\sigma_{\leq\tau}))$ is a GSMP consistent with the sample in $Path(\sigma_{\leq\tau})$. For all paths with prefix $\sigma_{\leq\tau}$ there exists a corresponding execution in the GSMP that produces the same path. Now, we introduce the definition (3) of stable equivalence relation that establish the similarity between states. This relation that is applied statistically allows the creation of a more abstract model from a set of paths $Path(\sigma_{\leq\tau})$.[4]

[4] The size of model at each equivalence iteration between states is reduced.

Definition 3. *Let $\mathcal{M} = (\mathcal{X}, \mathcal{E}, \Gamma, \boldsymbol{p}, p_0, \boldsymbol{G})$ be a stochastic automaton, a relation $R \subseteq \mathcal{X} \times \mathcal{X}$ is said to be a* stable relation *if and only if any s, s' have the following three properties,*

$$|\Gamma(s)| = |\Gamma(s')| \tag{8}$$

there is a one to one correspondence f between $\Gamma(s)$ and $\Gamma(s')$,

$$\begin{gathered} \text{if } \exists e \in \mathcal{E} \text{ and } \exists\, n \in \mathcal{X} \text{ such that } \boldsymbol{p}(n; s, e) > 0, \text{ then} \\ \exists\, n' \in \mathcal{X} \text{ such that } \boldsymbol{p}(n'; s', f(e)) > 0, \\ \boldsymbol{G}(s, e) \sim \boldsymbol{G}(s', f(e)), \text{ and } (n, n') \in R \end{gathered} \tag{9}$$

and

$$\begin{gathered} \text{if } \exists e \in \mathcal{E} \text{ and } \exists n, n' \in \mathcal{X} \text{ such that } n \neq n' \ , \\ \boldsymbol{p}(n; s, e) > 0 \text{ and } \boldsymbol{p}(n'; s, e) > 0 \text{ then} \\ \boldsymbol{p}(n; s, e) \sim \boldsymbol{p}(n; s', e) \text{ and } \boldsymbol{p}(n'; s, e) \sim \boldsymbol{p}(n'; s', e) \end{gathered} \tag{10}$$

where $|\Gamma(s)|$ is the number of feasible events in the state s, $\boldsymbol{p}$ is a probabilistic transition function, and $\boldsymbol{G}$ is a probability distribution function. Two states s and s' of $\mathcal{M}$ are said equivalent $s \equiv s'$ if and only if there is a stable relation R such that $(s, s') \in R$.

With the application of the definition (2), a loop transition can be produced when two states are merged in a GSMP. We need to ensure also the correctness of the state merge algorithm in order to know that the merged states are the same, otherwise we run the risk that the established equalities between states produce the model of an erroneous manner. So, a convergence analysis of our method is crucial as we will see in the next section.

We show now a concrete example for the application of the definition (3). For instance, assuming that have $|\Gamma(s)| = |\Gamma(s')| = 2$, $\Gamma(s) = \{a, b\}$, and $\Gamma(s') = \{c, d\}$. The equation (8) is trivially satisfied, i.e., the feasible event set have the same size. However, the equation (9) and (10) is not trivially satisfied. To be satisfied we need to conclude that $\boldsymbol{G}(s, a) \sim \boldsymbol{G}(s', c)$ and $\boldsymbol{G}(s, b) \sim \boldsymbol{G}(s', d)$, or $\boldsymbol{G}(s, a) \sim \boldsymbol{G}(s', d)$ and $\boldsymbol{G}(s, b) \sim \boldsymbol{G}(s', c)$ is true, if $\boldsymbol{G}(s, a) \sim \boldsymbol{G}(s, b)$, $\boldsymbol{G}(s, a) \sim \boldsymbol{G}(s', c)$ or $\boldsymbol{G}(s, a) \sim \boldsymbol{G}(s', d)$ then $\boldsymbol{p}(n; s, a) \sim \boldsymbol{p}(n'; s', b)$, $\boldsymbol{p}(n; s, a) \sim \boldsymbol{p}(n'; s', c)$, $\boldsymbol{p}(n'''; s, a) \sim \boldsymbol{p}(n'''; s', d)$ respectively, otherwise a Bernoulli test[5] is not necessary, and all states reachable by s and all states reachable by s' must also be a stable relation, i.e., the next states of (s, s') also have to satisfy these three properties.[6]

An existence of equal feasible event sets ($\Gamma(s) = \Gamma(s')$) created a non deterministic choice when merged. This problem can be solved applying a deterministic merge function, as follows:

$$\text{While } \exists s, x \in \mathcal{Q} \text{ and } \exists e \in E \text{ such as } s', s'' \in \sigma(s, x\, e),\ merge(s', s'') \tag{11}$$

The merge shall be made recursively until no more non-deterministic event transitions occur. We describe with a brief example the application of the equation 11. Let $\tau(s, x\, \nu(s_0)) = \{e\}$ and $\tau(s, x\, \nu(s'_0)) = \{e\}$ be two non-deterministic transitions from s_1 and s_2 labeled with same event e, respectively. Assuming that s_0 is merged in s'_0, we get a new non-deterministic choice between s_1 and s'_1 until to the end of the paths. Therefore, we need to apply the merge recursively until there are only deterministic choices. If the remaining paths are equal then one of them is removed.

[5] To test if two distributions have the same probability or follows the same Bernoulli distribution.

[6] In the definition (3), the real event identifiers are not necessary but we need to know that the sets of feasible events have associated for each event the same clock distribution.

Algorithm 1: Scheduler estimator (SE)

input : A $Path(\sigma_{\leq\tau})$ and a $Pt(Path(\sigma_{\leq\tau})) = (\mathcal{F}, \mathcal{Q}, \rho, \varrho, \delta)$.
output: The $Pt(Path(\sigma_{\leq\tau}))$ with replaced old clocks by original values of clocks.

```
for n ← 1 to |Path(σ≤τ)| do                                          // For all paths n
    for l ← 2 to |σ^n| do                                            // For all nodes l of path n
        for p ← l to 1 do                                            // Decrement p
            if ¬(σ^n‖ε[s, l] ∈ τ(ν(σ^n‖χ[s, p])) ∧ |τ(ν(σ^n‖χ[s, p]))| > 1 ∧
            σ^n‖ε[s, p] ≠ σ^n‖ε[s, l]) then
                p ← p + 1;
                break;
        if σ^n‖χ[s, p] ≠ σ^n‖χ[s, l] then
            Val ← 0;
            for t ← p to l do                                        // Estimating
                Val ← Val + σ^n‖_G[s, t];
                if σ^n‖χ[s, t] = σ^n‖χ[s, l then break;
            replace(Pr(Path(σ≤τ)), ν(σ^n‖χ[s, l]), Val);
```

3.2 Inferring the state age structure

The considered stochastic process, the GSMP, requires a state age memory structure. This state age memory, normally identified as a scheduler, allows the use of different continuous distributions to govern the inter-event times, i.e., the inter-event times between events of a GSMP might not be distributed with same distribution. It is not true in continuous-time Markov processes where the inter-event times follow always an exponential distribution. Nevertheless, the event scheduler is a data structure that allows to simulate the next event to be triggered at each step.

We introduce the notion of scheduler estimation in order to calculate the historic clock values for each event. Thus, we reconstruct values sampled from *new clocks* to estimate the events distribution of the model that produces those executions. For instance, supposing that have two events a and b that can be triggered in a state s_0, where s_0 is the initial state of the model, and there are two random variables $X_a \sim E(0.2)$ and $X_b \sim W(1, 0.1)$ associated to each event. The events a and b begin labeled as *new clock* and therefore two samples are given by random variables, respectively, X_a and X_b. Given the samples $x_a = 1.2$ and $x_b = 0.5$ from their respective distributions, the event b wins automatically. Next, the clock value of event b is subtracted and is stored with the value $1.2 - 0.5 = 0.7$ and a new clock is sampled to b. Then, the event a wins with value 0.7 versus the event b with new clock 1.4. We may calculate, therefore, the original value of the event a from the produced sample execution $\{s_0, (b, 0.5), s_1, (a, 0.7), \cdot\}$ summing inter-event times between a and b, such that $0.5+0.7 = 1.2$. Thus, we may say that the value sampled in state s_0 for the event a has the value 1.2, what is true.

Although the above presented scheme can be extended recursively to any finite sample execution, we need to clearly identify the *new clocks* and *old clocks* for any path. Suitable for that, we present the *scheduler estimator* algorithm which reconstructs the clock history structure. However, only the new clock samples are suitable to predict the distributions associated to each event in order to check the definition (9) as we will see later. The estimation process happens due to the existence of a relationship (i.e., a map function ν) between sample executions $Path(\sigma_{\leq\tau})$ and the prefix tree $Pt(Path(\sigma_{\leq\tau}))$ that is constructed from the same sample executions. Informally, we say that the map function ν associates a piecewise-path and a prefix tree node. The algorithm 1 has a particular order notation upon a set of paths $Path(\sigma_{\leq\tau})$ with prefix $\sigma_{\leq\tau}$ that is described as follows:

. σ^n is the n^{th} path $Path(\sigma_{\leq\tau})$, where $0 < n \leq |Path(\sigma_{\leq\tau})|$, and
. $\sigma^{n,l}$ is the l^{th} piecewise of path n, where $0 < l \leq |\sigma^n|$,

Algorithm 2: Probabilistic similarity of states (PSS)

input : A $Pt(Path(\sigma_{\leq\tau})) = (\mathcal{F}, \mathcal{Q}, \rho, \varrho, \delta)$, and a type I error α between $[0; 1]$.
output: A stochastic timed automaton $\mathcal{M}$.

```
M = PSA(Pt(Path(σ_{≤τ}))) ;                                   // See definition (2)
attempt ← 1;
while attempt > 0 do
    attempt ← 0;
    C ← clusterize(M);
    for n ← 1 to |C| do
        for k ← 1 to |C^n| do
            x ← k + 1;
            while C^{n,x} ≠ C^{n,|C^n|} do
                if is_active(C^{n,x}) then
                    if similar(C^{n,k}, C^{n,x}, α) then
                        dmerge(M, C^{n,k}, C^{n,x}, ·, ·);
                        inactivate(C^{n,x});
                        attempt ← attempt + 1;
                x ← x + 1;
```

where a variable between two symbols ('|') denotes its size. We explain, in the following, how the algorithm 1 estimates the original sampled clock values. First, the algorithm begins by traversing each path of sample executions set in a bottom-up order to know if the current event can be triggered by a clock with a label *new clock* or an *old clock*. In this step, we know that an old clock is valid when the successor nodes have this event activated, otherwise it is labeled as *inactive clock*. The algorithm goes to the predecessor node of the current node recursively, always in one sample execution, until we have encountered a possible inactive clock. When an inactive clock is encountered for the current event, in state $s^{\cdot}$, this implies that this event e cannot in $\tau(s^{\cdot})$, which is an active event set for a state $s^{\cdot}$. Therefore, in the worst case the first state (s_0) of the sample execution may be encountered. Given this element we can reconstruct the original clock value by the sum of the values between the found state ($s^{\cdot}$ or s_0) and the current state. Lastly, we replace the old clock value by the estimated original clock value.

3.3 Testing the similarity of states statistically

A prefix tree is given with reconstructed samples from the previous SE algorithm. Thus, a test to know that two distinct states in a GSMP are similar is needed. If there are, we need to merge them in only one state (a state merge paradigm). The state similarity estimation allows to test if both states are equal, merge them, and construct a deterministic stochastic timed automaton. The algorithm 2 allied with dependent functions represented by algorithms 3, 4 and 5 establishes a new methodology to learn GSMPs, which are processes that hold a semi-Markov property. We call the presented solution, in a more abstract manner, as *model identification*.

The probabilistic similarity of states algorithm. The algorithm 2 have notations associated to the ordered set of clusters and also between these cluster elements, as follows:

. the set of n ordered clusters C, classified by events, are denoted by C^n, and
. $C^{n,k}$ is the k^{th} element of cluster C^n, for each $1 \leq n \leq |C|$ and $1 \leq k \leq |C^n|$.

Algorithm 3: `similar` function

input : Two states s_1 and s_2, and a type I error α.
output: Boolean, *true* if it is similar, or otherwise *false*.

$\Gamma_1 \leftarrow \tau(s_1)$; $\Gamma_2 \leftarrow \tau(s_2)$;
if $|\Gamma_1| \neq |\Gamma_2|$ **then** return *false*;
for each e_1 **in** Γ_1 **do**
  while $|\Gamma_2| > 0$ **do**
    $e_2 \leftarrow \texttt{get}(\Gamma_2)$;
    $F_{n_1} = \mathcal{T}(\varrho(s_1 e_1))$; $F_{n_2} = \mathcal{T}(\varrho(s_2 e_2))$;
    if $\sqrt{\frac{n_1 n_2}{n_1+n_2}} \sup_x |F_{n_1}(x) - F_{n_2}(x)| > K_\alpha$ **then**
      if $\texttt{similar}(\delta(s_1 e_1), \delta(s_2 e_2), \alpha) \neq true$ **then**
        return *false*;
      continue;
    $\texttt{put}(\Gamma_2, e_2)$;
for each e_1, e_2 **in** Γ_1 **such that** $s_1\, e_1 \sim s_1\, e_2$ **do**
  if $|\varrho(s_1\, e_1) - \varrho(s_1\, e_2)| > \sqrt{\frac{1}{2} log \frac{2}{\alpha}} \left(\frac{1}{\sqrt{n_1}} + \frac{1}{\sqrt{n_2}} \right)$ **then**
    return *false*;
if $|\Gamma_2| < 1$ **then** return *true*; **else** return *false*;

The clustering function `clusterize` produces groups of elements C with a selection based on the feasible event set $\tau(s_\cdot)$ for each state $s_\cdot$ of $\mathcal{M}$, where $\mathcal{M}$ at first attempt is equal to $\texttt{PSA}(Pt(Path(\sigma_{\leq\tau})))$. The `is_active` and `inactivate` functions allow that only the prefix tree nodes that were not merged are used, and the function `similar` test the similarity between two feasible event sets $\tau(C^{n,k})$ and $\tau(C^{n,x})$.

The PSS algorithm shall be subdivided in three blocks. The first block collects the clusters of states with same feasible event set (given by the equation 8) until there is no more equivalent states (i.e., while $\texttt{attempt} > 0$) in an iterative process, thereby implies that the feasible event set of prefix tree nodes may change along the execution (change made by the `dmerge` function).[7] The second block tests the similarity of two states applying the third block if the states are both similar. Thus, we use the `similar` function defined in algorithm 3 that uses the Kolmogorov-Smirnov test [6, p. 552] to decide if two empirical probabilistic distributions are equal or not. In general, it verifies through a statistical equivalence whether there are a one to one correspondence of events between two feasible event sets satisfying thereby the equation 9. The third block consists in a merge process, where our algorithm 2 merges the equal states by the function `dmerge`. It allows the construction of the stochastic timed automaton as well as the solution to the problem of non-deterministic state merge, i.e., when two states have the same set of events.

The function `similar` *(algorithm 3).* The similarity between two feasible event sets Γ_1 and Γ_2 within the type I error α is solved by the algorithm 3. Thus, the Kolmogorov-Smirnov test (K-S test) [6, p. 552] is applied to test if two distributions are or are not the same (i.e., compare two empirical cumulative distribution functions). Let $\{X_n\}_{n\geq 1}$ and $\{Y_n\}_{n\geq 1}$ be two independent successions of

[7] A performance comparison between our algorithm and the others cited in the state of the art [1,11] may be made. We conclude that their computational structure is quite similar, but the performance of our method is better due to a selection method based in clusters, which reduces drastically the number of comparisons. Our method can be implemented as a parallel algorithm to increase the performance and to support more complex systems.

independent real random variables with common distribution functions, respectively F_1 and F_2. The K-S test allows testing two hypothesis,

$$H_0 : F_1(x) = F_2(x), \text{ for all } x \in \mathbb{R} \text{ against} \qquad (12)$$
$$H_1 : F_1(x) \neq F_2(x), \text{ for some } x \in \mathbb{R}$$

using the statistic test,

$$T_{n_1,n_2} = \sqrt{\frac{n_1 n_2}{n_1 + n_2}} \sup_{x \in \mathbb{R}} |F_{n_1}(x) - F_{n_2}(x)| \qquad (13)$$

where F_{n_1} and F_{n_2} denotes respectively the empirical distribution functions associated to the samples $(X_1, ..., X_{n_1})$ and $(Y_1, ..., Y_{n_2})$. The random variable T_{n_1,n_2} converges to the Kolmogorov distribution whose values are tabled in [6, p. 555]. For a significance level α, we reject H_0 when the observed value $\widehat{T}_{n_1,n_2}$ of the test statistic for the particular samples $(x_1, ..., x_{n_1})$ and $(y_1, ..., y_{n_2})$ exceeds the value K_α, with $G(k_\alpha) = 1 - \alpha$. The two empirical cumulative distributions F_{n_1} and F_{n_2} are estimated using the function $\mathcal{T}$. This function estimates the empirical distribution from a set of sample clocks as follows:

$$\mathcal{T}_n(x) = \frac{\textit{clock value of } z_1, z_2, ..., z_n \textit{ that are} \leq x}{N} \qquad (14)$$

where x is the threshold of the cumulative function, and z_i for all events $i \in D$ and $D \subseteq \mathcal{E}$ are the sample clock values.

We now describe the algorithm 3. This algorithm begins by the comparison of two feasible event sets Γ_1 and Γ_2. The comparison shall be established by a one to one relation between events in feasible sets. If the relationship between events is complete then the states are similar and so it allows the check of equation 9. However, there is another particularity in this algorithm that is when two events have the same 'id' in the feasible event set, for two states respectively. For instance, this indicate that the event is triggered as e but there are different probabilities in transition probability matrix. To solve this, we construct a hypothesis test for two Bernoulli distributions using Hoeffding bounds in order to know if the occurrence probabilities are the same (i.e., verifies the equation 10). This method is the same to that in [11]. The method checks if the means $\varrho(s_1\, e_1)$ and $\varrho(s_1\, e_2)$ of two Bernoulli distributions are statistically different or not.[8]

The function `dmerge` *(algorithm 4).* The non-deterministic choices caused by the state merge technique (as produced by the algorithm 2) shall be solved with the function `dmerge`, described in the algorithm 4. This means that two paths starting at the same event are merged recursively to only one path. The recursivity of function `dmerge` is defined by three stop clauses or base cases, as follows:

1. two states s_1 and s_2 are equal,
2. the state s_2 does not have any active event in their feasible event set, and
3. the state s_1 or state s_2 is merged in previous steps of recursion.

Otherwise, a recursive call is always made, characterized by a set of rules which reduce all other cases toward the base case.

The data structures *tree* and *heap* are used in algorithm 4 as auxiliaries data structures. The `pt_get`, `pt_find`, `pt_add`, and `pt_remove` are functions that describe the classical operations in a auxiliary prefix tree, as name indicates to get the equivalent state that was merged, find, add event sequences (states) and remove event sequences (states). The `heap_put` and `heap_get` also are functions that describes a classical heap operations.

A heuristic shall be applied in the function `dmerge`. This allows that the state merge converges into the pretended model more quickly. The heuristic is based on the degree of the states, therefore,

[8] Note that the `get` and `set` functions are both known functions in *set* theory. `get` retrieve a element from a set, and `set` put an element in a set.

Algorithm 4: `d_merge` function

input : A stochastic timed automaton $\mathcal{M}$, two states s_1 and s_2, a heap h, and a prefix tree t.
output: A deterministic stochastic timed automaton $\mathcal{M}$.

if $degree[s_1] > degree[s_2]$ **then** `state_swap`$(\mathcal{M}, s_1, s_2)$;
if $s_1 = s_2$ **then** return; `// The recursive stop condition;`
if `pt_find`(t, s_1) *or* `pt_find`(t, s_2) **then**
    `heap_put`$(h, (s_1, s_2))$;
    return;
$removed[s_2] = True$; `// Set the state` s_2 `as removed in` $\mathcal{M}$`;`
`pt_add`(t, s_1); `pt_add`(t, s_2); `// Add` s_1 `and` s_2 `states to the prefix tree` t`;`
Update states from $\mathcal{M}$ that have a transition pointing to state s_2 for state s_1;
if $|\tau(s, s_2)| > 0$ **then** `// There exists at least one active event from` s_2`;`
    for each *event* e **in** $\tau(s, s_2)$ **do**
        if $e \in \tau(s, s_1)$ **then**
            `d_merge`$(\mathcal{M}, s_1\, e, s_2\, e, h, t)$;
        else
            `psa_insert`$(\mathcal{M}, s_1, e)$;
`pt_remove`(t, s_1); `pt_remove`(t, s_2); `// Remove` s_1 `and` s_2 `states from` t`;`
while $count[h] > 0$ **do**
    $tuple \leftarrow$ `heap_get`(h);
    if `pt_find`(t, s_1) *or* `pt_find`(t, s_2) **then** *break*;
    if $removed[first[tuple]]$ **then**
        $tuple =$ (`pt_get_merged`$(first[tuple]), second[tuple])$;
    if $removed[first[tuple]] = False$ *and* $removed[second[tuple]] = False$ **then**
        `d_merge`$(\mathcal{M}, first[tuple], second[tuple], h, t)$;

the states with lower degree are merged in state with higher degree, which indicates that the state with lower degree is removed. Then, every state that is removed from the deterministic stochastic timed automaton $\mathcal{M}$ is labeled as removed, and s_1 and s_2 are added to another prefix tree t in order to avoid merge them more than once time in the recursion path. The `pt_remove` indicates that the states can be merged in the following recursion paths. This allows to block the state merge when occurs a state that depends of the previous state that is merging, i.e., occurs normally when there is a loop transition between the previous state s_1 and the new state s_2 which depends of s_1. In addiction, an updating for the states in $\mathcal{M}$ that point to removed state s_1 are needed. The function `pt_get_merged` allows their update by returning the new pointers of similar states (i.e., that were merged) at the current recursivity step. Lastly, if there is at least one element in the heap h then merge the pair of states which cannot be merged in the previous steps. This solves the recursive merge dependence.

3.4 Inferring event distributions using maximum likelihood

To conclude the presentation of the learning method, we need to introduce the concept of distribution discriminant and its selection criteria. Given a prefix tree without similar states, we need to estimate the parameters for each event distribution (i.e., an empirical distribution) that better fits the sample data. It should therefore be used the maximum likelihood estimator (MLE) to estimate the distribution parameters and a selection criteria as the maximum log likelihood to select the better distribution with properly estimated parameters [7].

We describe the algorithm 5 that estimates the distribution parameters using the maximum likelihood estimator (MLE) for continuous distributions such as: Exponential, Weibull and Log-

Algorithm 5: Estimation function

input : A deterministic stochastic timed automaton $\mathcal{M}$.

output: A deterministic stochastic timed automaton $\mathcal{M}$ with associated random variables and those distributions.

for each n **in** Q **such that** $removed[n] = 0$ **do**
  for each e **in** $\tau(s, n)$ **do**
    $G_e \leftarrow \int_0^\infty arg \max_{f_d \in \mathcal{D}} \{ln\left[\mathcal{L}_d(\varrho[n\, e])\right]\}$;

Normal. However, there are other continuous distributions, like Rayleigh, Normal (without negative part), that we are not described in detail in this report, but that can be applied upon this algorithm. The MLE from a distribution d is defined by the maximization of the likelihood function defined by $\prod_{i=0}^{n} f_d\left(x_i \mid \theta\right)$ regarding the set of parameters θ. The selection criteria (described in detail in [7]) is based in log likelihood which may be calculated by the expression,

$$ln\left[\mathcal{L}_d\left(\theta \mid x_1, ..., x_n\right)\right] = \sum_{i=0}^{n} ln\left[f_d\left(x_i \mid \theta\right)\right] \tag{15}$$

where $\mathcal{L}_d$ is the maximized likelihood function, θ is the set of estimated parameters for a distribution f_d where $\mathcal{L}_d$ is maximized, and $x_1, ..., x_n$ are samples to be estimated (in this case, they are values of clocks associated to an event that follows a specific distribution to be estimated). This selection criteria is defined by the maximum value of the calculated log likelihood, i.e.,

$$ln\left[\mathcal{L}_{dm}\right] > max\ \{\forall d \in \mathcal{D} \text{ s.t. } d \neq dm \text{ then } ln\left[\mathcal{L}_d\right]\} \tag{16}$$

where D is a set of distributions in analysis, and $ln\left[\mathcal{L}_d\right]$ the log likelihood of distribution d. The distribution with maximum likelihood is denoted by $dm \in \mathcal{D}$. Thus, two or more distributions are needed to make a decision with this criteria.[9] In addition, this method estimates the distribution that, in the limit, will be equal to the distribution that produces these samples to learning.

4 Model identification in the limit

In order to guarantee the correct model identification, we have to show that the GSMP that our learning algorithm produces is similar to the model that was used to generate the samples. We describe the mandatory requirements to the correct model identification, composed by the following three clauses:

1. the prefix tree constructed by sample executions provided by a GSMP, $Pt(Path(\sigma_{\leq\tau}))$, is also a GSMP,
2. the sample executions to learn have the minimum information to form the model, and
3. the application of a state merge upon $Pt(Path(\sigma_{\leq\tau}))$, in the limit, converges to one similar model that identifies the $Path(\sigma_{\leq\tau})$.

Indeed, the correctness of the learning algorithm depends essentially of the correctness of the state merge procedure, since that the definition 2 is correct by construction and a structurally complete sample to learn is assumed. This means that with the definition 2 we guarantee the first clause, and that we should focus attention in the other two remaining clauses. For the second clause, we need to

[9] The distributions of set $\mathcal{D}$ are distributions with a parameter or a set of parameters estimated by the MLE method previously.

ensure that the sample executions to learn forms a *structurally complete sample* (SCS). This is known as the problem of insufficient data training and when this occurs is obviously impossible to learn the model that produces incomplete sample executions. For the third clause, we need to ensure that, in the limit, the error of merge two non equivalent states tends to zero.[10] Assuming these three clauses satisfied, we prove that the model that is learned by our learning algorithm, in the limit, behaves as the original stochastic system.

4.1 Ensure a structurally complete sample

The common methods to achieve a structurally complete sample like reachability analysis are not enough when the model is not known. In this case acquire a SCS is really a great challenge and it is not trivial. It turn, the selection of termination probability for a sample execution that allows to achieve a SCS upon unknown models. Moreover, the probability measure of a path from a unknown model is not trivially ensured.

A SCS is a sample composed by a set of paths that explores every feasible transition and every reachable state. This structure solves a common problem known as *insufficient data training* to learn a model, i.e., only with paths of infinite size is guaranteed that for any model, the learned model eventually converges to an equivalent. With a SCS, we ensure that the minimum information needed to learn a model from sample executions is achieved. In order to ensure that a set of paths relying on SCS, we introduce a termination probability p_t as a solution. The simulation technique is described, as follows:

1. simulate the SDES M,
2. terminate when probability measure of a path $\sigma_{\leq\tau}$ of execution is less than p_t, i.e., $\mu(C(\sigma_{\leq\tau}, \langle E_k, Y_k^* \rangle, X_k, ..., \langle E_n, Y_n^* \rangle, X_n)) < p_t$, and
3. apply recursively the steps 1. and 2. to generate more sample executions.

We simply note that the solution method based on termination probability has weaker correctness guarantees than reachability analysis. It also places a greater responsibility on the user, who has to choose a good value for p_t. The automatic achievement of p_t is not trivial, and it is out of the scope of this report.

4.2 The state merge error, in the limit, converges to zero

Assuming the checking of the two first clauses, our learning algorithm can only make errors when testing the similarity between two states. In addiction, the errors α and β between two event distributions of the K-S test are defined, as follows:

. α is the type I error of H_0 be rejected, where in fact H_0 should not be rejected, and
. β is the type II error of H_1 be accepted, where in fact H_1 should be rejected.

This means that the state merging errors α_s and β_s are defined by the multiplication of the errors made by the distribution comparison of each event $\alpha_s = \prod_{i=1}^{k} \alpha_i$ and $\beta_s = \prod_{i=1}^{k} \beta_i$, where k is the number of similar events. Moreover, the model errors α^* and β^* are equal to the multiplication of the error α_s and β_s used for each state merged $\alpha^* = \prod_{i=1}^{n} \alpha_s[i]$ and $\beta^* = \prod_{i=1}^{n} \beta_s[i]$, where n is the number of merged states.

We present, in the following, two propositions about the bounds of type II error.

Proposition 1. *Suppose the Kolmogorov-Smirnov test for two samples with size n_1 e n_2 respectively, and a significance level α. For sufficiently large samples, i.e., when $n_1 \to \infty$ and $n_2 \to \infty$, β tends to zero.*

[10] The error that does not merge two equivalent states is guaranteed by the Kolmogorov-Smirnov test.

In the following we present a sketch of the proof. The proof of this proposition is based on the following facts: by the theorem of Glivenko-Cantelli when H_0 is true and n_1 and n_2 tend to infinity, $\sup_{x\in\mathbb{R}} |F_{n_1}(x) - F_{n_2}(x)|$ converges certainly to zero. So, from the uniqueness of the limit, when H_0 is true and $n_1 \to \infty$, $n_2 \to \infty$, we have that

$$\sqrt{\frac{n_1 n_2}{n_1 + n_2}} \sup_{x\in\mathbb{R}} |F_{n_1}(x) - F_{n_2}(x)| \tag{17}$$

tends certainly to $+\infty$. Therefore, in the validity of H_1, the probability of rejecting H_0 tends to 1, which was to be demonstrated.

We know that the convergence of k-S test is exponential. Moreover, more details about the proposition that we present here and also β error boundaries can be seen in [22] and [12].

Proposition 2. *Supposing that we assume the type II error β, in the limit, for the K-S test convergent to zero, a multiplication of the type II error $\prod_{i=1}^{k} \beta_i$, in the limit, also tends to zero.*

This proposition is trivially satisfied. Given the limit law of multiplication, we know that the $\lim_{x\to a} f(x)\cdot g(x) = \lim_{x\to a} f(x)\cdot \lim_{x\to a} g(x)$. Thus, due to $f(x) = g(x)$, then the limit is maintained.

5 Tool and proof of concept

The presented learning algorithm was implemented in a tool that allows the learning and the analysis of a set of case studies, such as: task schedulers, land mobile satellite communication systems, and network traffic model estimation. The practical example that is showed in this report is the learning and analysis of a scheduler for a multi-processor.

5.1 SDES toolbox for Matlab

We have developed a SDES toolbox[11] that applies the presented learning algorithm. The toolbox was developed to analyze and learning GSMPs. It also supports the model description by a event-driven language that can be directly used as input model language to a GSMP model checker [19]. We also developed a graphical user interface in order to simplify and make a user friendly interface. The SDES toolbox also have command line interface.

The framework was developed in C and C++ language. These languages are interconnected with the Matlab engine and the interface functions are called in the Matlab console using the known Matlab language. The GUI is created with Qt toolkit for binding C++ language.

5.2 Stochastic analysis of a scheduler for a multi-processor

An optimal scheduler design for a multi-processor system with uncertainty in tasks duration is difficult to achieve and is clearly a great challenge [15]. In the figure 1, we present the model from which is possible to achieve statistically answers about the worst case sequence and the optimal case sequence, in particular, of a two-processors scheduler system. There are in this system two processors where we can run two tasks at same time. Supposing that there are three tasks $\{a, b, c\}$, we can only run two tasks at same time and the other one when one of the tasks is done. Thus, the presented model have eleven states that describe the state of the two processors at time, and the tasks that were done. The scheduler can make three choices, begins with (a, b), (a, c), or (b, c). The event *init* of the model, representing these choices: $p([, ab, c]; [, , abc], init) = \frac{1}{3}$, $p([, ac, b]; [, , abc], init) = \frac{1}{3}$, and $p([, bc, a]; [, , abc], init) = \frac{1}{3}$ respectively. These choices bounds the time (i.e., worst and optimal) of the

[11] Available from http://desframework.sourceforge.net/

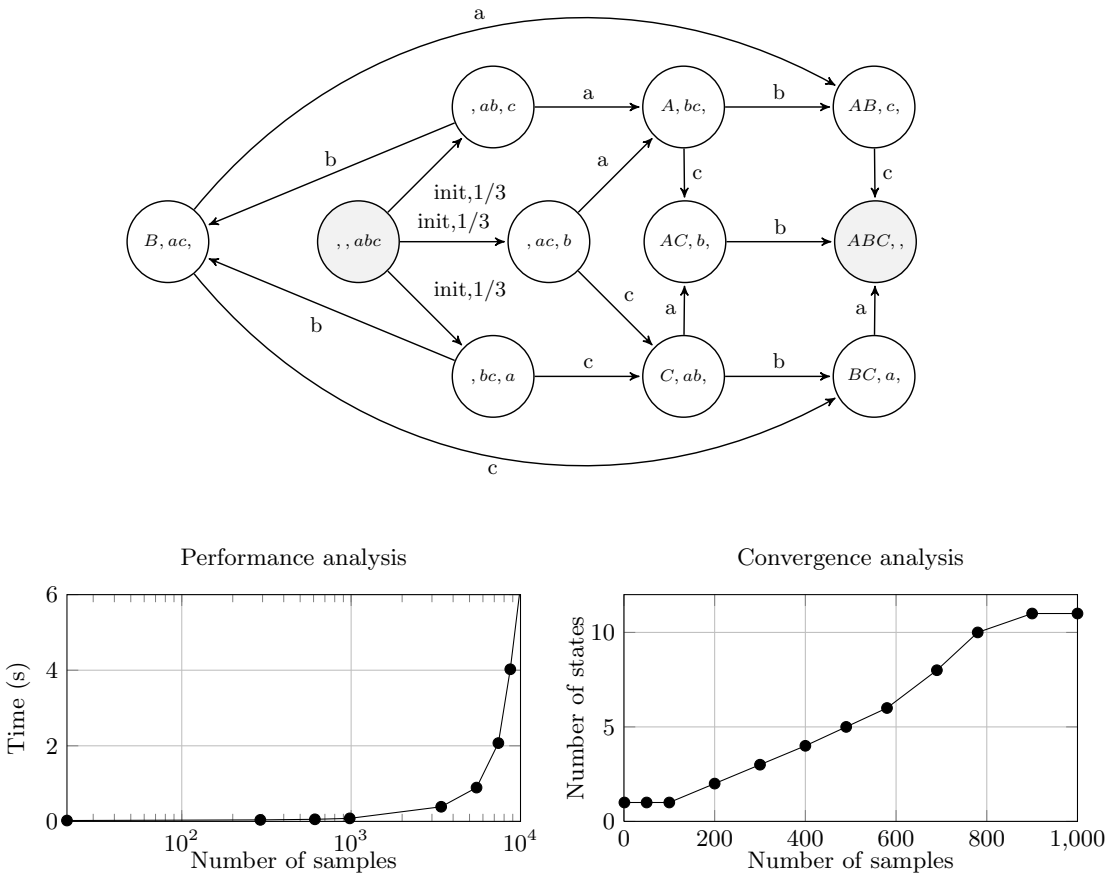

Fig. 1. Learning GSMP of a multi-processor scheduler with uncertainty

execution for these three tasks. If we have a scheduler that is completely random (i.e., the probability of events $\{ab, ac, bc\}$ are equiprobable) where the probability for each choice is equal then we select the path with maximum probability which means that is the better sequence. Thus, if we have a scheduler that begins with the optimal tasks then we have a optimal scheduler for these tasks. However, we need to describe two situations, as follows: if we use only exponential distributions the chose is easy, the rate of distribution identifies the order (lower expected value is the more probable), but otherwise if have different continuous distributions the ordering selection is not so trivial. This is the case where our method solves. For instance, given $init : T_{init} \sim Exponential(1)$, $a : T_a \sim Weibull(0.1, 1)$, $b : T_b \sim Exponential(0.4)$, and $c : T_c \sim Log\text{-}Normal(0, 0.25)$, respectively.

Given the sample executions that form a SCS, we have compared the performance and the convergence of our algorithm when sample executions grow, on the figure 1. We can see in this convergence graph that for one thousand sample executions, the model converges into a model with same number of states. According to the correctness of our learning algorithm, we can ensure that if we grows infinitely the model converges certainly to the original model. However, we verify, in this example, that the learned model with our algorithm acquired with approximately nine hundred sample executions have the same event language of the original model. This process was made in one machine with an *Intel Core 2 Duo CPU T7500 @ 2.2Ghz* that have *4Gb* of memory.

An interesting point in this model is that the path with more probability to occur is the optimal case execution and the path with lower probability is the worst case execution, when we have a random scheduler.

6 Conclusion and Future Work

We have presented a novel learning algorithm to learning GSMPs from deployed stochastic systems for which we do not know the model before-hand. The learn algorithm can be used to verify the deployed systems using existing probabilistic model-checking tools. We also have developed a toolbox for Matlab that applies the techniques described in this report. We also show with our experiment, a scheduler analysis for multi-processor system, that stochastic automata are really capable and scalable. We can use our algorithm for analysis of a stochastic system but also to verify or testing it. However, one of the limitations of our work is that it may not scale up for systems having large stochastic automata. Development of techniques that allows the approximate verification as the model is learning may be the solution.

Acknowledgments

We would like to thank to Ana Paula Martins by the very constructive discussions about the statistical properties of the proposed learning algorithm. We also thank to Paul Andrew Crocker by the revision of some topics in this report, and lastly we thank to Kouamana Bousson and Thierry Brouard by the initial answers about stochastic systems and Markov chains.

References

1. Rafael C. Carrasco and Jose Oncina. Learning deterministic regular grammars from stochastic samples in polynomial time. *RAIRO (Theoretical Informatics and Applications*, 33:1–20, 1999.
2. Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
3. Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikucionis, and Zheng Wang. Time for statistical model checking of real-time systems. In *CAV*, pages 349–355, 2011.
4. André de Matos Pedro. Learning and testing stochastic discrete event systems, 2011.
5. André de Matos Pedro, Maria João Frade, Ana Paula Martins, and Simão Melo de Sousa. Learning generalized semi-Markov processes: From stochastic discrete event systems to testing and verification. *INForum2011 - SOFTPT*, pages 160–165, 2011.
6. Morris H. DeGroot. *Probability and Statistics*. Addison Wesley, 2nd edition, 1989.
7. Arabin Kumar Dey and Debasis Kundu. Discriminating among the log-normal, weibull, and generalized exponential distributions. *IEEE Transactions on Reliability*, 58(3):416–424, 2009.
8. P. W. Glynn. A gsmp formalism for discrete event systems. *Proceedings of The IEEE*, 77:14–23, 1989.
9. E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.
10. Mor Harchol-Balter and Allen B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Trans. Comput. Syst.*, 15:253–285, August 1997.
11. Christopher Kermorvant and Pierre Dupont. Stochastic grammatical inference with multinomial tests. In *Proceedings of the 6th International Colloquium on Grammatical Inference: Algorithms and Applications*, ICGI '02, pages 149–160, London, UK, UK, 2002. Springer-Verlag.
12. Jerome Klotz. Asymptotic efficiency of the two sample Kolmogorov-Smirnov test. *Journal of the American Statistical Association*, 62(319):932–938, 1967.
13. Axel Legay, Benoît Delahaye, and Saddek Bensalem. Statistical model checking: An overview. In *RV*, pages 122–135, 2010.
14. Ming-Wei Lu and Cheng Julius Wang. Weibull data analysis with few or no failures. In Hoang Pham, editor, *Recent Advances in Reliability and Quality in Design*, pages 201–210. Springer London, 2008.

15. Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 3rd edition, 2008.
16. Koushik Sen, Mahesh Viswanathan, and Gul Agha. Learning continuous time markov chains from sample executions. In *Proceedings of the The Quantitative Evaluation of Systems, First International Conference*, pages 146–155, Washington, DC, USA, 2004. IEEE Computer Society.
17. William J. Stewart. *Probability, Markov Chains, Queues, and Simulation: The Mathematical Basis of Performance Modeling*. Princeton University Press, Princeton, NJ, USA, 2009.
18. Wei Wei, Bing Wang, and Don Towsley. Continuous-time hidden Markov models for network performance evaluation. *Perform. Eval.*, 49:129–146, September 2002.
19. Håkan L. S. Younes. Ymer: A statistical model checker. In *CAV*, pages 429–433, 2005.
20. Håkan L. S. Younes, Edmund M. Clarke, and Paolo Zuliani. Statistical verification of probabilistic properties with unbounded until. In *SBMF*, pages 144–160, 2010.
21. Hakan Lorens Samir Younes. *Verification and planning for stochastic processes with asynchronous events*. PhD thesis, Pittsburgh, PA, USA, 2004.
22. C. S. Yu. Pitman efficiencies of Kolmogorov-Smirnov test. *The Annals of Mathematical Statistics*, 42(5):1595–1605, 1971.