Learning and testing stochastic discrete event systems

André de Matos Pedro

January 2012

School of Engineering
University of Minho
Braga, Portugal

Supervisors
Simão Melo de Sousa
Maria João Frade

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Computer Engineering

This research was sponsored by the EVOLVE project with reference BI6-2010_EVOLVE_UMINHO.

Abstract

Discrete event systems (DES) are an important subclass of systems (in systems theory). They have been used, particularly in industry, to analyze and model a wide variety of real systems, such as production systems, computer systems, traffic systems, and hybrid systems. Our work explores an extension of DES with an emphasis on stochastic processes, commonly called *stochastic discrete event systems* (SDES). There was a need to establish a stochastic abstraction for SDES through a *generalized semi-Markov processes* (GSMP). Thus, the aim of our work is to propose a methodology and a set of algorithms for GSMP learning, using model checking techniques for verification, and to propose two new approaches for testing DES and SDES (non-stochastically and stochastically). This work also introduces a notion of modeling, analysis, and verification of continuous systems and disturbance models in the context of verifiable *statistical model checking*.

Acknowledgments

Foremost, I would like to express my sincere gratitude to my supervisor Prof. Simão Melo de Sousa for the continuous support of my MSc study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better supervisor and mentor for my MSc study.

I would like to express also my sincere gratitude to my second supervisor Prof. Maria João Frade, for her encouragement, insightful comments, and hard questions.

My sincere thanks also go to Prof. Ana Paula Martins, Prof. Paul Andrew Crocker, Prof. Kouamana Bousson, and Prof. Thierry Brouard, by very constructive talks and very interesting explanations.

I thank my fellow labmates of the RELEASE, NMCG, and SOCIALAB: Angelo Arrifano, Silvio Filipe, Gil Santos, João Vasco, Mário Pereira, Luis Pedro, and Rui Pedro by the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the lab in the last year. Also I thank my friends in University of Minho.

In particular, I am grateful to Dr. João Paulo Patricio for enlightening me the first glance of hard programing at the high school.

Last but not the least, I would like to thank my family: my parents José Pedro and Eugénia Pedro, for giving birth to me at the first place and supporting me spiritually throughout my life, and to my brother Eduardo.

In the past few years, however, the person closest to my heart has been my beloved Sara. Her love is the inspiration for my professional achievements.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Discrete event systems (DES) are actually a wide class of systems that are encountered daily. The canonical example is a simple queuing system with a single server. For example, consider a post office with only one employer, and therefore one queue for letters, packages, etc. The customers arrive 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. There is no synchronization between the arrival and departure of customers, i.e. the two events just introduced are asynchronous and do not occur at the same time so this is clearly an example of a DES. Coping with asynchronous events is the major advantage of DES. Other examples of DES besides queuing systems include, computer systems, communication systems, manufacturing systems, traffic systems, database systems, software systems - telephony, and hybrid systems (hybrids between DES and *continuous-variable dynamic systems* (CVDS)).

This thesis explores an extension of DES with an emphasis on stochastic processes. This type of system is commonly called a *stochastic discrete event system* (SDES). When we talk about stochastic processes, we have to know that the dynamics of DES is described by a set of random variables (i.e., a stochastic process). For example, the arrival rate of customers at the post office or the risk that the post office is closed for some reason. These random variables have certain probabilistic distributions that can model the system behavior. We concern ourselves with how these probability distributions are obtained. These distributions are given by a collection of realistic measures to which we fit an analytic distribution function. This will be made due to a well established learning algorithm that accurately captures the timing of events from the real world, therefore there are no empirical assumptions.

1.1 The Problems

In this thesis, we consider one main problem and three secondary problems. The main problem is the learning of *generalized semi-Markov processes* (GSMP). The next problem is to establish a relation between GSMP and SDES in order to verify them. The second secondary problem is to establish abstractions for continuous systems in order to be modeled as DES or SDES (respectively, non-stochastic and stochastically). The last problem is to apply stochastic and non-stochastic testing generation for the SDES such as perturbation model or classical unit testing suite. In the following lines, we shall describe these four problems in a more detailed manner.

- The learning of well defined stochastic processes has always been a great challenge. Although there exist complex models that are analytically intractable, actually there is a set of problems that were solved by the technique of model checking (more precisely statistical model checking). However, existing solutions assume that the acquisition of empirical models is enough. Today, we could safely say that they are not enough for most real systems like critical systems, due to its growing complexity. In this case, for the verification, only a model and a set of properties are needed to be able to determine whether the system satisfies some given property. Thus, verifying empirical models may not make sense. Solving the learning problem for a class of stochastic processes allows us to verify not only the model but to ensure some guaranties about the implementation and allows us to use that model for test generation.

- The relation between SDES and GSMP must be compatible (coincident). Thus, the GSMP has to be established as a more abstract model for SDES. With this, we can verify statistically SDES.

- Continuous variable dynamic systems are a class of systems that include all known dynamic systems defined by differential or difference equations. As we know, this type of systems require in many cases a lot of computational resources. At present, continuous systems run in discrete event platforms like a computer, for example, a satellite calculates its perturbations and corrections to maintain its trajectory. So, if a continuous system is executed as a discrete system, we can model it with some abstractions as a DES. There are many advantages. First, we can simulate systems with much less resources than those that are needed for continuous cases. Second, we can simplify (with abstractions) with a minor loss of precision. This is a vast problem that would require another new thesis. In this thesis we will address only

the preliminaries of the problem by defining and identifying the challenges to be addressed. Third, we can use DES easily to make its verification through the model checking technique and for test generation. Fourth, we can also use it for generation of perturbation models, for dynamic systems. Lastly, there exists a lot of simulators for discrete event systems, including parallel simulation.

– Testing is an essential methodology to target some parts of one system to find a set of problems/defects (bugs). The original challenge of this thesis was to derive a new methodology for test generation from stochastic models. The complex systems are not traceable to test all possible inputs. So, model testing generation requires a set of realistic models, otherwise we are running the risk of create many tests that are not time feasible (time-consuming and potentially expensive procedures). We shall discuss, in the next chapters, the problem of generating excessive test sets. A method to reduce it is using test generation based on realistic inputs, some that occurs in a realistic scenario shall also be discussed in this thesis.

1.2 Summary of research contribution

SDES are a large class of known stochastic systems with a good intuitive basis. At present, there are several learning algorithms that can be adopted for SDES. For instance, Sen et al. [2004b] has proposed learning the continuous-time Markov processes, but other processes or algorithms for a more general approach do not exist. However, there is no learning algorithm for stochastic systems that does not hold the Markov property. The generalized semi-Markov processes are a large class of stochastic processes that does not hold it. Obviously, this made this type of model more complex and analytically intractable. Hence, learning GSMP is statistically an ambitious due to its potentialities in this area.

We propose in this thesis a new and unique methodology for learning generalized semi-Markov processes that is the most extensive model when lifetimes can be governed by any continuous probabilistic distributions. As we know from classical Markov processes, the exponential distributions are not enough to model the lifetime of a product (e.g., a electronic component life) [Lu and Wang, 2008] or even a computer process [Harchol-Balter and Downey, 1997, Leland and Ott, 1986].

We propose an algorithm to learn generalized semi-Markov processes, that can be used for stochastic discrete event systems. We show with our experiments that this type of model is really capable and scalable. We can use it for analysis of an industrial system but also to verify or testing it.

We would like to highlight the Stochastic DES toolbox for Matlab, in particular, that

have come out of our research efforts and is now available to the public [1].

1.3 Overview of this thesis

This thesis is concerned with discrete event systems, in particular on stochastic extensions. Furthermore, it introduces the concept of a perturbation model for continuous systems. A comprehensive introduction to terminology, notation, and techniques that are used extensively throughout the thesis is given in chapter 2. It contains a brief overview on DES, stochastic processes, and SDES. Moreover, it presents some known common examples.

Chapter 3 provides the context for our research contribution with a discussion of related work in learning generalized semi-Markov processes, abstractions for continuous systems using the *discrete event specification* (DEVS) approach, probabilistic/statistical model checking, and test generation for stochastic processes.

We start in chapter 4 by introducing the notion of learning generalized semi-Markov processes. Some preliminary definitions are made in order to understand the terminologies that are made there. This work originated in an effort to develop a learning approach for SDES [de Matos Pedro et al., 2011]. We also describe the notion of abstraction for continuous systems and two approaches for statistical model-base test generation.

The evaluation of the learning algorithm is described in chapter 5. Three case studies are studied. The first, consists on an empirical evaluation of the generalized-semi Markov processes learning algorithm. The second, a realistic case study for the analysis of availability from a communication between a satellite and a high-speed train. Lastly, an abstraction of a continuous system is made in order to be described and simulated as DES.

Finally, chapter 6 discusses directions for future work in continuous systems abstractions and stochastic testing. For abstractions of continuous systems there is a need for applying this approach to stochastic polynomials or polynomial chaos. For testing it shall be adopted a full stochastic approach for creating a toolbox to test generation for SDES.

[1] http://sourceforge.net/projects/desframework/

Chapter 2

Systems and models

In this chapter some basic concepts of system's theory based on the definitions of Cassandras and Lafortune [2006] and Zimmermann [2007] shall be given. No detailed explanations about the concepts of *state space*, *sample equation*, *sample path* and *feedback* shall be given. We suppose that the reader is familiarized with those concepts, nevertheless, it should be suficient for the reader to use their intuition in order to understand them.

We begin our description of *discrete event systems* (DES's) by first identifying their fundamental characteristics, and by presenting a few familiar examples of such systems. Lastly, we describe two extensions (deterministic and stochastic) of DES's that will be used in the next chapters.

2.1 An introduction to discrete event systems

We shall now describe *system* in terms of a primitive concept like a set or a mapping. We provide three definitions about "What is a system" as found in Cassandras and Lafortune [2006]:

- An aggregation or assemblage of things so combined by nature or man as to form an integral or complex whole.
- A regularly interacting or interdependent group of items forming a unified whole.
- A combination of components that act together to perform a function not possible with any of the individual parts.

In these definitions there are two salient features. First, a system is composed of a set of components, and second a system is associated with one function that presumably intends to perform something.

DES's are a particular class of systems that are largely used in industry as pattern model Cassandras and Lafortune [2006]. So, when the state space of a system is naturally described by a discrete set like $\{0, 1, 2, ...\}$, and state transitions are only observed at

discrete points in time, we associate these state transitions with *events* and talk about a DES.

Usually DES's can be used as an hybrid model combined with *continuous-variable dynamic systems* (CVDS). The CVDS changes with measured quantities such as temperature, pressure and acceleration, which are continuous variables evolving over time. It is usually characterized by a continuous state space. So, in contrast to CVDS, DES's evolves in a discrete state space, where at each transition there exists an associated event (i.e., an event-driven system).

2.1.1 The concept of event

The event is a primitive concept with a good intuitive basis. As Cassandras and Lafortune [2006] say we need to "... emphasize that an event should be thought of as occurring instantaneously and causing transitions from one state value to another.".

An event can be identified when a specific action occurs (e.g., a package arrived at a warehouse or a button is pressed), or it can be viewed as a spontaneous occurrence dictated by nature (e.g., a computer crash for whatever reason too complicated to figure out or a set of sensors that are unstable when temperature changes), or it can be viewed as a result of several conditions which are suddenly all met (e.g, the fluid level in a tank has just exceeded a given value or a warehouse that suddenly is full).

At this point, it is clear for us that DES's are event-driven systems, they depend on discrete occurrences (i.e., something occurs due to an action, an event or something not well defined). Thus, we will introduce the informal definition of these systems and compare it with continuous systems which are guided by the time.

Event-driven systems and time-driven systems. In continuous-state systems the state generally changes as time changes. This is particularly evident in discrete-time models, where the *clock* is what drives a typical *sample path*. With every *clock tick* the state is expected to change, since continuous state variables continuously change with time. It is because of this property that we refer to such systems as time-driven systems. In this case, the time variable (t in continuous time or k in discrete time) is a natural independent variable which appears as the argument of all input, state, and output functions.

In discrete-state systems, we say that the state changes only at certain points in time through instantaneous transitions. To each such transition we can associate an event. However, the description of the timing mechanisms based on which events take place before it can be triggered is missing. We describe it in detail in section 2.2.

2.1.2 The model characterization and its abstractions

The event-driven property of DES was discussed in the previous section. It refers to the fact that the state can only change at discrete points in time, which physically correspond to occurrences of asynchronously generated discrete events. From a modeling point of view, this has the following implication. If we can identify a set of *events* any one of which can cause a state transition, then time no longer serves the purpose of driving such a system and may no longer be an appropriate independent variable. Thus, the DES satisfies the following two properties:

1. The state space is a discrete set.
2. The state transition mechanism is event-driven.

However, according to these properties and as described by Cassandras and Lafortune [2006] DES's are informally described by the definition 1. So, the *state space* is some discrete set $X = \{s_1, s_2, ..., s_n\}$, where X is a set of finite states that have the state s_1, s_2, ..., s_n and n is a finite size of state space (see definition 2).

Definition 1. A Discrete Event System (DES) is a discrete-state, event-driven system, that is, its state evolution depends entirely on the occurrence of asynchronous discrete events over time.

Definition 2. The state space of a system, usually denoted by X, is the set of all possible states that the system may take.

The *sample path*[1] can only transiting from one state to another whenever an event occurs. Note that an event may take place, but not cause a state transition. For example when one event occurs at state s_2 and goes to the same s_2 state (i.e., an arc or a loop transition). It is often convenient to represent a DES *sample path* as a timing diagram (timeline) with events identified by vertical dashed lines at the times they occur, and states shown in between events like the figure 2.3.

DES's have a set of models available for the most diverse systems. So, due to this, we describe below some abstractions for different DES's. Note that DES's may be viewed as an event-language or even just a language.

The abstractions of DES's. Languages, timed languages, and stochastic timed languages represent the three levels of abstraction at which DES's are modeled and studied: untimed (or logical), timed, and stochastic. We describe in the following sections (see

[1]The *sample path* is one simulation (sample execution) of a DES; It identifies the behavior of DES's over time, where the time instants are discrete steps in the state space.

sections 2.2 and 2.4) the timed automaton and stochastic timed automaton, which are two abstract models for describing DES's.

Remark 2.1. One should not confuse discrete event systems with discrete-time systems. The class of discrete-time systems contains both CVDS and DES's. In other words, a DES may be modeled in continuous or in discrete time, just like a CVDS can.

2.1.3 Hybrid models

Systems that combine time-driven with event-driven dynamics are referred to as hybrid systems. Recent years have seen a proliferation of hybrid systems largely due to the embedding of microprocessors (operating in event-driven mode) in complex automated environments with time-driven dynamics; examples arise in automobiles, aircraft, chemical processes, heating, ventilation, and air-conditioning units in large buildings, etc.

We do not describe here formally any definition for a hybrid model, therefore, we will try to deduce one perturbation model from a continuous variable system based on other abstractions such as differential equations and difference equations.

A more close example of a hybrid system will be shown later in section 4.4, where a perturbation model from an inverted pendulum is described and modeled.

2.1.4 Examples

The subtleties of DES's were described in the previous sections. There are many common examples such as: a computer system crashing due to periodical bugs that obligate its watchdog to reboot it or even a bottle filling line of an industry (a hybrid industrial system). Here we shall discuss in detail one practical example of a warehouse system manager.

The model that we will describe is quite misunderstood from a mathematics point of view principally due to discontinuities (there is more time when nothing occurs than when sometimes does occurs). Now, in this sense what we need to explain is that there are other models with better abstractions, which we will explain in the next sections. They are timed automata (see section 2.2) and stochastic timed automata (see section 2.4).

Example 2.2. Consider a warehouse containing packages of goods from a shipping company (a particular case of the example provided by Cassandras and Lafortune [2006]). When a new package is received by the shipping company, this represents an arrival event at the warehouse and when the shipping company dispatches the package it is a departure event. So, a truck periodically delivers and loads up a certain number of products, which are thought of as departures from the warehouse (figure 2.1). With this model we can

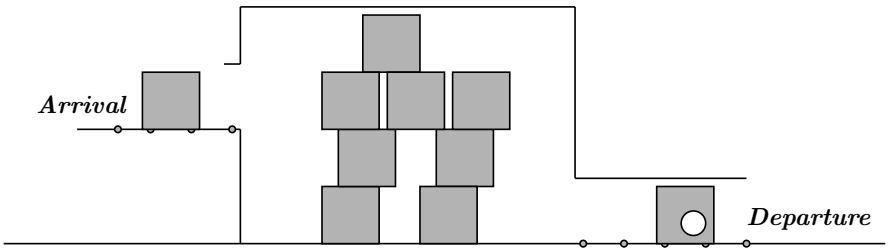


Figure 2.1: A warehouse of a shipping company depicted in a simple diagram of input of goods and the output of packages. The packages arrive at the warehouse as *arrival* events and the departures of packages from warehouse as *departure* events. Thus, the DES is defined by the number of packages at the warehouse (the state space) and the actions triggered by this two events.

check if the storage capacity of a particular warehouse located in Lisbon is enough or not. We can analyze how many packages have arrived and departed at one time instant of the system or also over a long run execution.

We can analyze the warehouse system as a queuing system. This system can be modeled as a DES where the number of packages is determined by the state space of the system. Thus, we define $x(t)$ to be the number of products at time t, and define an output equation for our model to be $y(t) = x(t)$. For instance supposing that at most ten packages can be stored, we will have a system with discrete state space $X = \{1, 2, 3, 4, ..., 10\}$.

We define two binary functions u_1 and u_2 that correspond to the arrival event and departure event to indicate that in time t an event has occurred,

$$u_1(t) = \begin{cases} 1 & \text{If a package arrives at time } t \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

and

$$u_2(t) = \begin{cases} 1 & \text{If a package departs at time } t \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

The sequence of time instants when $u_1(t) = 1$ defines the *schedule* of package arrivals at the warehouse. Similarly, the sequence of time instants when $u_2(t) = 1$ defines the *schedule* of package departures from the warehouse.

We next describe the simplifications that we have assumed for the model. First, the warehouse have space for at most of ten packages and its storage capacity is reached at ten packages. Next, the loading of the truck takes zero time. Then, the truck can only take away a single product at a time. Lastly, a package arrival and a package departure never take place at exactly the same instant, that is, there is no t such that $u_1(t) = u_2(t) = 1$. In order to derive a state equation for this model, let us examine all possible state transitions we can think of:

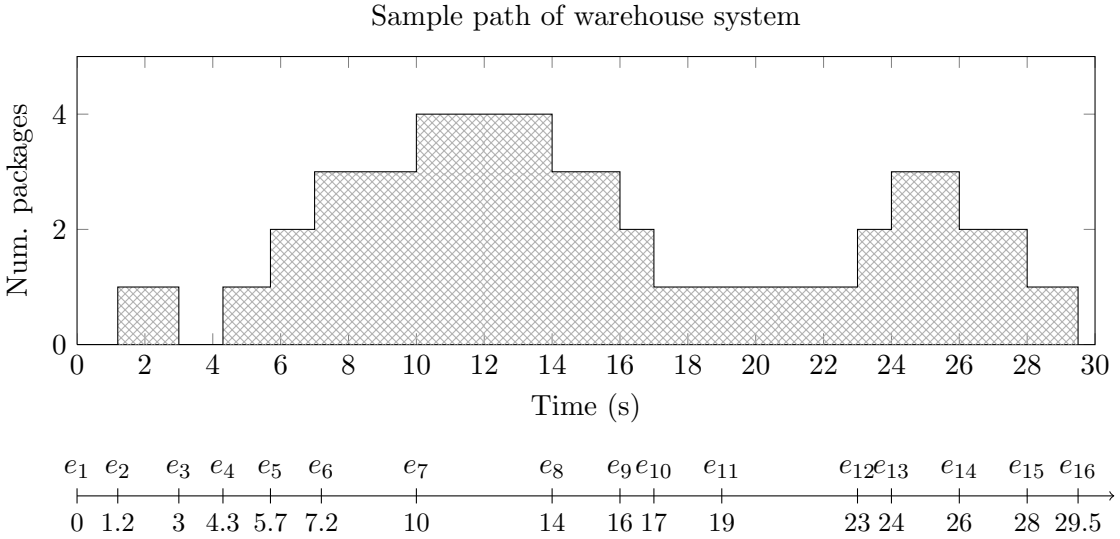


Figure 2.2: A *sample path* of the warehouse system with an event timeline on bottom.

1. $u_1(t) = 1$, $u_2(t) = 0$. This simply indicates an arrival at the warehouse at time instant t. As a result, x(t) should experience a jump of +1, if $x(t) \leq 10$.

2. $u_1(t) = 0$, $u_2(t) = 1$. This means a truck is present at time t, and we should reduce the warehouse content by 1. However, there are two sub-cases. If $x(t) > 0$, then the state change is indeed -1. But if $x(t) = 0$, the truck finds the warehouse empty and the state does not change.

3. $u_1(t) = 0$, $u_2(t) = 0$. Clearly, no change occurs at time t.

Let t^+ denote the time instant *just after* t. With this notation, based on the observations above we can describe it with the following state equation:

$$x(t^+) = \begin{cases} x(t) + 1 & \text{if } (u_1(t) = 1,\ u_2(t) = 0,\ x(t) \leq 10) \\ x(t) - 1 & \text{if}(u_1(t) = 0,\ u_2(t) = 1,\ x(t) > 0) \\ x(t) & \text{otherwise} \end{cases} \tag{2.3}$$

A typical *sample path* of this system is shown in figure 2.2. In this case, $u_1(t) = 1$ (i.e., packages arrive) at time instants t_1, t_2, t_3, t_5, t_6, t_{12}, and t_{13}, and $u_2(t) = 1$ (i.e., a truck arrives) at time instants t_4, t_7, t_8, t_9, t_{10}, and t_{11}. Note that even though a truck arrival takes place at time t_{11}, the state $x(t) = 0$ does not change, in accordance with figure 2.2.

Remark 2.3. The equation 2.3 is an illustration of many situations we could be encountering on the common DES. This warehouse system is modeled as a stack, therefore the mathematical model is rather informal and it is not the most interesting model. In most of the time nothing happens due to the discontinuities of the functions u_1 and u_2.

2.2 Timed automata

An automaton is a model that is capable of representing a language (i.e., a sequence of input symbols) according to well-defined rules. The simplest way to present the notion of automaton is to consider its representation as a directed graph, or state transition diagram. So, we present here the definitions of the *deterministic timed automaton* (DTA) followed by the definition of the *clock structure*. Lastly, we expose a practical example of a simple DTA.

Definition 3. The *deterministic timed automaton* is a six-tuple $M = (\mathcal{X}, \mathcal{E}, f, \Gamma, x_0, V)$, where

$\mathcal{X}$	is a countable state space,
$\mathcal{E}$	is a countable event set,
$f : \mathcal{X} \times \mathcal{E} \to \mathcal{X}$	is a state transition function and is generally a partial function on its domain,
$\Gamma : \mathcal{X} \to 2^{\mathcal{E}}$	is the active event function (or feasible event function); $\Gamma(x)$ is the set of all events e for which $f(x, e)$ is defined and it is called the active event set (or feasible event set),
x_0	is the initial state, and
$V = \{v_i : i \in \mathcal{E}\}$	is a clock structure.

Having presented the definition of DTA we can make some remarks about it, however we focus here only on the essential. Our remarks (see Cassandras and Lafortune [2006]) are:

- The functions f and Γ are completely described by the state transition diagram of the automaton.
- The automaton is said to be deterministic because f is a function from $\mathcal{X} \times \mathcal{E}$ to $\mathcal{X}$, namely, there cannot be two transitions with the same event label out of a state.
- The fact that we allow the transition function f to be partially defined over its domain $\mathcal{X} \times \mathcal{E}$ is a variation over the usual definition of automaton in computer science literature that is quite important in DES's theory.
- Formally speaking, the inclusion of Γ in the definition of M is superfluous in the sense that Γ is derived from f. One of the reasons why we care about the contents of $\Gamma(x)$ for state x is to help distinguish between events e that are feasible at x but cause no state transition, that is, $f(x, e) = x$, and events e' that are not feasible at x, that is, $f(x, e')$ is not defined.
- The event set $\mathcal{E}$ includes all events that appear as transition labels in the state transition diagram of automaton M. In general, the set $\mathcal{E}$ might also include additional events, since it is a parameter in the definition of M. In other words, this can be composed by a parallel composition.

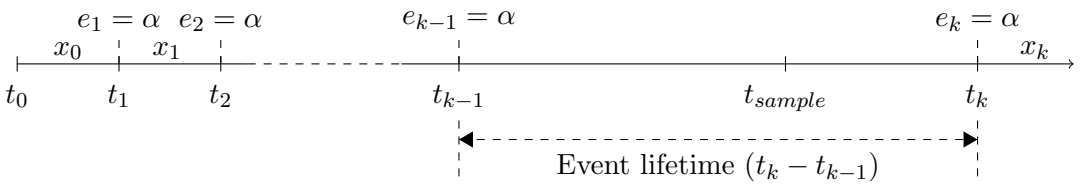


Figure 2.3: The timeline of a simple DES with one α event is depicted. The lifetime of an event of one DES is the time between two consecutive identical events, in this example, around t_{sample} have a lifetime of $t_k - t_{k-1}$.

– The V clock structure has one clock associated for each event. Each clock is made by a composition of an ordered set of fixed clock values. We formally describe it in the following section.

2.2.1 The clock structure

We introduce here the key ideas of timed DES's. As we have previously seen a simplified DES is a system composed of a finite set of states $\mathcal{X}$, and a finite set of events $\mathcal{E}$. It has one transition function f, that given an event knows what is the next state and an activation function Γ that knows what events are active in a given state. So, knowing all of this a clock structure is needed in order to drive the time of each event.

Example 2.4. (A DES with a single α event.) Let $\mathcal{E} = \{\alpha\}$, and $\Gamma(x) = \{\alpha\}$ for all $x \in \mathcal{X}$. A simulation of this system always produces the same output. So, the generated path, an event sequence, is denoted by $p = (e_1, e_2, e_3, ..., e_n),\ e_k = \alpha,$ for each $k = \{1, ..., n\}$. The time instant associated with the k^{th} event is denoted by $t_k,\ k = \{1, 2, ..., n\}$. The length of the time interval defined by two successive occurrences of the event is called a lifetime as we can see in the timeline of the figure 2.3. We denote v_k, the lifetime of the k^{th} event. Thus, we define $v_k = t_k - t_{k-1} \in \mathbb{R}^+$, for each k.

The evolution of this system over time can be described as follows. At time t_{k-1}, the k^{th} event is said to be activated or enabled, and is given a lifetime v_k. A clock associated with the event is immediately set at the value specified by v_k, and then it starts ticking down to 0. During this time interval, the k^{th} event is said to be active. The clock reaches zero when the lifetime expires at time $t_k = t_{k-1} + v_k$. At this point, the event has to occur. This will cause a state transition. The process then repeats with the $(k+1)^{th}$ event becoming active.

To introduce some further notation, let t_{sample} be any time instant, not necessarily associated with an event occurrence. Suppose that $t_{k-1} \leq t \leq t_k$. Then t divides the interval $[t_{k-1}, t_k]$ into two parts (see figure 2.3) such that $y_k = t_k - t$ is called the clock

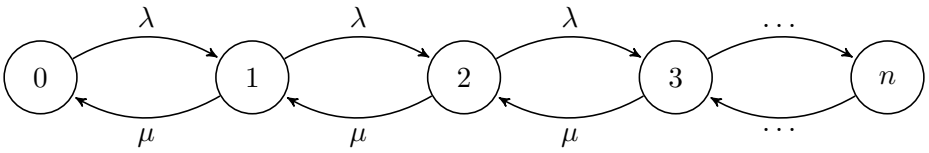


Figure 2.4: A discrete event system of the D/D/1/n stack.

or residual lifetime of the k^{th} event, and $z_k = t - t_{k-1}$ is called the age of the k^{th} event. It is obvious that $v_k = z_k + y_k$. It should be clear that a *sample path* of this DES is completely specified by the lifetime sequence $(v_1, v_2, ..., v_n)$. This is also referred to as the clock sequence of event α.

Example 2.5. (A simple timed automaton - a stack model with n elements.) This model is a more interesting DES. It has concurrence between two events. Let the event set be $\mathcal{E} = \{\lambda, \mu\}$. The state transition diagram of the figure 2.4 depicts a stack model with n states. The set of nodes is the state set of the automation, $\mathcal{X} = \{0, 1, 2, 3, ..., n\}$. The labels of the transitions are elements of the event set (alphabet) $\mathcal{E}$ of the automaton. The arcs in the graph provide a graphical representation of the transition function of the automaton, which we denote as $f : \mathcal{X} \times \mathcal{E} \to \mathcal{X}$:

$$\begin{aligned} f(0, \lambda) &= 1 \\ f(1, \lambda) &= 2,\ f(1, \mu) = 0 \\ f(2, \lambda) &= 3,\ f(2, \mu) = 1 \\ &... \\ f(n, \mu) &= n - 1 \end{aligned}$$

The notation $f(1, \lambda) = 2$ means that if the automaton is in state 1, then upon the occurrence of event λ, the automaton will make an instantaneous transition to state 1. The cause of the occurrence of event λ is irrelevant; the event could be an external input to the system modeled by the automaton, or it could be an event spontaneously generated by the system modeled by the automaton. So, if model is in state 1 that has $f(1, \lambda) = 2$ and $f(1, \mu) = 0$. At this point the system can trigger one of those events. These events compete with each other one to trigger the event with less holding time. For instance, if $v_\lambda = (1, 10, 30, ...)$ and $v_\mu = (2, 5, 32, ...)$ we begin by making a transition λ (the only possible transition) with time $v_{\lambda,1}$, and next compare the clock value of $v_{\lambda,2}$ and $v_{\mu,1}$ when μ event wins. Then, the automaton will make an instantaneous transition to state 1 and trigger the event λ with a holding time of $v_{\lambda,2}$, and so on. It should be clear that a *sample path* of this stack model has the event time sequence $(1, 2, 10, 5, 30, ...)$.

2.3 Stochastic process basics

In this section, we briefly introduce the concept of stochastic process always followed by practical examples.

A *stochastic process* is simply a collection of random variables indexed through some parameter, which is normally thought of as time. For example, suppose that $\Omega = \{IDLE, CRASHED\}$ is the sample space for describing the random state of a computer system (like the last one). By mapping IDLE into 0 and CRASHED into 1 we may define a random variable $X(\omega), \omega \in \Omega$, which takes the values 0 or 1. Thus, $X(\omega) = 1$ means the computer system is CRASHED, and $X(\omega) = 0$ means the computer system is IDLE. Next, suppose that we describe the state of the computer system over time as discrete steps. Let $k = 1, 2, 3, ...$ be the index of the time (i.e., seconds, hours, days, etc...). Thus, $X(\omega, k)$ is a random variable describing the state of computer system on the k^{th} time step. The collection of random variable $\{X(\omega, 1), X(\omega, 2), ..., X(\omega, k), ...\}$ defines a stochastic process.

Definition 4. A *stochastic or random process* $X(\omega, t)$ is a collection of random variables indexed by t. The random variables are defined over a common probability space $(\omega, \mathbb{E}, P)$ with $\omega \in \Omega$. The variable t ranges over some given set $\mathbb{T} \subseteq \mathbb{R}$.

The probability space is defined by the 3-tuple $(\omega, \mathbb{E}, P)$, where ω determine the sample space, which is the set of all possible outcomes, $\mathbb{E}$ is a set of events, where each event is a set containing zero or more outcomes, and P the probabilities of the events.

The mathematical abstractions for DES's with stochastic clock structure (see later) is a *stochastic process*, which is formally a collection of random variables (as we have seen above) that we denote by $\{X(t) \mid t \in \mathbb{T}\}$, where t is the time indexing. The parameter t may have a different interpretation in other environments. Index set $\mathbb{T}$ denotes the set of time instants of observation. The set of possible results of $X(t)$ is called *state space* of the process (a subset of $\mathbb{R}$), and each of its values corresponds to a state.

Stochastic processes are characterized by the state space and the index set $\mathbb{T}$. If the state space is discrete (countable), the states can be enumerated with natural numbers and the process is a *discrete-state process* or simply a chain. $\mathbb{T}$ is then usually taken as the set of natural numbers $\mathbb{N}$. Otherwise, it is called a *continuous-state process*. Depending on the index set $\mathbb{T}$ the process is considered to be *discrete-time* or *continuous-time*. Four combinations are obviously possible. In our setting of stochastic discrete event systems, we are interested in systems where the flow of time is continuous ($\mathbb{T} = \mathbb{R}_0^+$) and the state space is discrete. The stochastic process is thus a continuous-time chain and each $X(t)$ is a discrete random variable.

Now, we describe the inherent properties of a *Markov process*. They are the following two properties:

Property 2.6. the future state only depends on the current state and not on the states history (*no state memory is needed*)

Property 2.7. the inter-event time in the current state is irrelevant in determination of the next state (*no age memory is needed*)

A discrete-state stochastic process $\{X(t)|t \in \mathbb{T}\}$ is called a *Markov chain* if its future behavior at time t depends only on the state at t.

$$P\{X(t_{k+1}) = n_{k+1}|X(t_k) = n_k, ..., X(t_0) = n_0\} = P\{X(t_{k+1}) = n_{k+1}|X(t_k) = n_k\} \quad (2.4)$$

Processes that hold a Markov property are easier to analyze than more general processes because information about the past does not need to be considered for the future behavior. Markov chains can be considered in discrete or continuous time, and are then called *discrete-time Markov chain* (DTMC) or *continuous-time Markov chain* (CTMC). From the memoryless[2] property of a Markov process it immediately follows that all inter-event times must be exponentially (CTMC) or geometrically (DTMC) distributed. Different relaxations allow more general times. Examples are *semi-Markov processes* [Barbu and Limnios, 2008] with arbitrary distributions but solely state-dependent state transitions and *renewal processes* that count events with arbitrary but independent and identically distributed interevent times.

A *generalized semi-Markov process* (GSMP) [Glynn, 1989] allows arbitrary inter-event times like semi-Markov process. The Markov property of state transitions depending only on the current state is achieved by encoding the remaining delays of activities with nonmemoryless delay distributions in the state, which then has a discrete part (the system states) and a continuous part that accounts for the times of running activities.

2.4 Stochastic timed automata

Stochastic timed automata (STA's) according to the definition of timed automata have changes on clock structure and a little difference in the transition function. Broadly, the stochastic automaton is an automaton with stochastic clocks.

We begin by adopting a random variable notation to define the clock structure of timed automata as stochastic clocks: X is the current state; E is the most recent event (causing

[2]The memoryless is a property that some random distributions satisfies. The exponential distributions of non-negative real numbers and the geometric distributions of non-negative integers.

the transition into state X); T is the most recent event time (corresponding to event E); N_i is the current score of event i; and Y_i is the current clock value of event i. As in the deterministic case, we use the prime (') 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'.

Besides the stochastic clock structure specification, there are two additional probabilistic features:

- The initial automaton state x_0 may not be deterministically known. In general, we assume that the initial state is a random variable X_0. What we need to specify then is the *probability mass function* (pmf) of the initial state

$$p_0(x) = P[X_0 = x], \text{ where } x \in \mathcal{X} \tag{2.5}$$

- The state transition function f may not be deterministic. In general, we assume that if the current state is x and the triggering event is e', the next state x' is probabilistically specified through a transition probability[3]

$$p(x'; x, e') = P[X' = x' | X = x, E' = e'], \text{ where } x, x' \in \mathcal{X}, e' \in \mathcal{E} \tag{2.6}$$

A timed automaton equipped with a *stochastic clock structure* (see section 2.4.1), an initial state cumulative distribution function, and state transition probabilities defines a Stochastic Timed Automaton, as defined below.

Definition 5. A Stochastic Timed Automaton is a six-tuple $M = (\mathcal{X}, \mathcal{E}, \Gamma, p, p_0, G)$ where

$\mathcal{X}$	is a countable state space,
$\mathcal{E}$	is a countable 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* $P[X_0 = x], \ x \in \mathcal{X}$, of the initial state X_0, and
$G = \{G_i : i \in \mathcal{E}\}$	is a stochastic clock structure.

The automaton generates a stochastic state sequence $\{X_0, X_1, ...\}$ through a transition mechanism (based on observations $X = x$, $E' = e'$): $X' = x'$ with probability $p(x'; x, e')$ and it is driven by a stochastic event sequence $\{E_1, E_2, ...\}$ generated through

$$E' = arg \min_{i \in \Gamma(X)} \{Y_i\} \tag{2.7}$$

[3]Note that if $e' \notin \Gamma(x)$, then $p(x'; x, e') = 0, \forall x' \in \mathcal{X}$.

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

$$Y_i' = \begin{cases} Y_i - Y^* & \text{if } i \neq E' \text{ and } i \in \Gamma(X) \\ V_{i,N_i+1} & \text{if } i = E' \text{ or } i \notin \Gamma(X) \end{cases} \quad i \in \Gamma(X') \tag{2.8}$$

where the interevent time Y^* is defined as

$$Y^* = \min_{i \in \Gamma(x)} \{Y_i\} \tag{2.9}$$

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

$$N_i' = \begin{cases} N_i + 1 & \text{if } i = E' \text{ or } i \neq \Gamma(X') \\ N_i & \text{otherwise} \end{cases} \quad i \in \Gamma(X') \tag{2.10}$$

and

$$V_{i,k} \sim G_i \tag{2.11}$$

where the tilde ($\sim$) notation denotes "with distribution". In addition, initial conditions are: $X_0 \sim p_0(x)$, and $Y_i = V_{i,1}$ and $N_i = 1$ if $i \in \Gamma(X_0)$. If $i \notin \Gamma(X_0)$, Y_i is undefined and $N_i = 0$.

Having presented the formal definition of stochastic timed automaton we need to make some remarks about comparison with definitions of the timed automaton. The transition function f is replaced by the probabilistic transition function p (the transition is made with uncertainty), x_0 the initial state is replaced by p_0 the probabilistic mass function (the initial state is given by this function; it can start from a different set of states given by the *pmf*), and lastly the main difference is that the stochastic automaton have G the stochastic clock structure instead of the V simple structure of events lifetime.

2.4.1 The stochastic clock structure

The clock structure of the stochastic timed automaton is defined according to definition 6. The stochastic process generates the lifetime sequences for each event given by $V_{i,k}$. The clock structure of the timed automaton is the same of the used in this definition, but the difference is that it is governed by a stochastic process.

Definition 6. The *stochastic clock structure* (or timing structure) associated with an event set $\mathcal{E}$ is a set of distribution functions

$$G = \{G_i : i \in \mathcal{E}\} \tag{2.12}$$

characterizing the stochastic clock sequences

$$\{V_{i,k}\} = \{V_{i,1}, V_{i,2}, ...\},\ i \in \mathcal{E},\ V_{i,k} \in\ \mathbb{R}^+,\ k = 1, 2, ... \tag{2.13}$$

2.5 The stochastic discrete event system

A *stochastic discrete event system* (SDES), according to Zimmermann [2007], is a tuple $SDES = (SV, A, S, RV)$ describing the finite sets of *state variables* SV and *actions* A together with the sort function S. The *reward variables* RV correspond to the quantitative evaluation of the model.

- S is a function that associates an individual sort to each of the state variables SV and *action variables* $Vars$ in a model. The *sort* of a variable specifies the values that might be assigned to it. We will not specify here the type system, which we assume as implicit. We will present a simple example (see later) to understand better how the system behaves. The variable types and operations can be described by the semantics of the event language in the appendix B.
- SV is the finite set of n state variables, $SV = \{sv_1, ..., sv_n\}$, which is used to capture states of the SDES.
- A denotes the set of actions of a SDES model. They describe possible state changes of the modeled system. An action $a \in A$ of a SDES is composed of a set of attributes that we are not describe here (please consider Zimmermann [2007]).
- RV is the notion of a quantitative evaluation of SDES called as *reward variable*. It is used as a set of variables to the analysis of the SDES (i.e., performance analysis, produce documents for documentation purposes, and verification). This is one feature that can be coupled for any DES therefore we will not describe it here.

2.5.1 Inclusion between stochastic timed automata and discrete event systems

The SDES definition comprises state variables, actions, sorts, and reward variables.

$$SDES = (SV, A, S, RV) \tag{2.14}$$

In order to capture a stochastic automaton as defined in the previous section these variables are set in the following manner. There is exactly one state variable sv, whose value is the state of the automaton. The set of state variables SV has therefore only one element.

$$SV = \{sv\} \tag{2.15}$$

The set of SDES actions A is given by the events of the automaton.

$$A = \mathcal{E} \tag{2.16}$$

The sort function of the SDES maps the state variable to the allowed values, i.e., the state space of the automaton. As there are no SDES action variables necessary, no sort is defined for them.

$$S(sv) = \mathcal{X} \tag{2.17}$$

The set of all possible states in SDES and in STA is obviously equal. The condition function also is always true, because all states in $\mathcal{X}$ are allowed. The initial value of the state variable is directly given by the initial state of the automaton. The actions of the SDES correspond to events of the stochastic automaton.

The remaining concepts and details that are interesting but not relevant for this work are left to the reader. For more details see for instance Zimmermann [2007].

2.5.2 The generalized semi-Markov process

A stochastic timed automaton is used to generate the stochastic process $\{X(t)\}$. This stochastic process is referred to as a *generalized semi-Markov process* (GSMP).

Definition 7. A Generalized Semi-Markov Process (GSMP) is a stochastic process $\{X(t)\}$ with state space $\mathcal{X}$, generated by a stochastic timed automaton $(\mathcal{X}, \mathcal{E}, \Gamma, p, p_0, G)$.

The GSMP simulation

The simulation concepts introduced by Ross [2006] and Banks [1998] shall now be described as well as the algorithms. We focus in particular on the simulation of generalized semi-Markov processes [Asmussen and Glynn, 2007].

Let us see the following example in order to understand the concept of simulation of a GSMP, its scheduler behavior, and its event competition.

Example 2.8. Given the M/M/1/n stack model (notation according to Kendall [1953]) of figure 2.5 that has $\mathcal{E} = \{\lambda, \mu\}$, $\mathcal{X} = \{0, 1, 2, 3, ..., n\}$ the discrete state space, and $\Gamma(0) = \{\lambda\}, \Gamma(1) = \{\lambda, \mu\}, ..., \Gamma(n-1) = \{\lambda, \mu\}, \Gamma(n) = \{\mu\}$. As was described in the last sections, the behavior of the DES is that the events compete each other to trigger the event with the minimal time. So, with that in mind and looking to the timeline of the figure 2.6, that is one simulation of the stack model, we can view this event competition. The shadowing below the arrows is the more probable lifetime given by the distributions related to λ and μ events with high probability (black) and with low probability (gray). Notoriously, we can view after the first triggered event λ that at each discrete step the selection of the triggered events is the minor value of the two λ and μ events. The behavior is the same as DES but with stochastic clocks that have uncertainty in time duration.

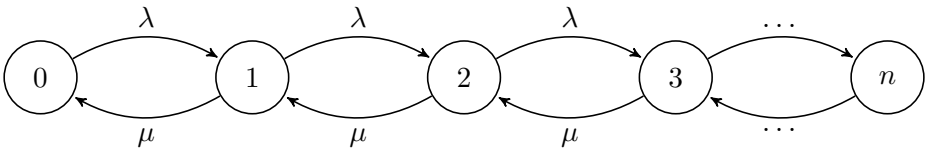


Figure 2.5: The stochastic discrete event system of the M/M/1/n stack is depicted.

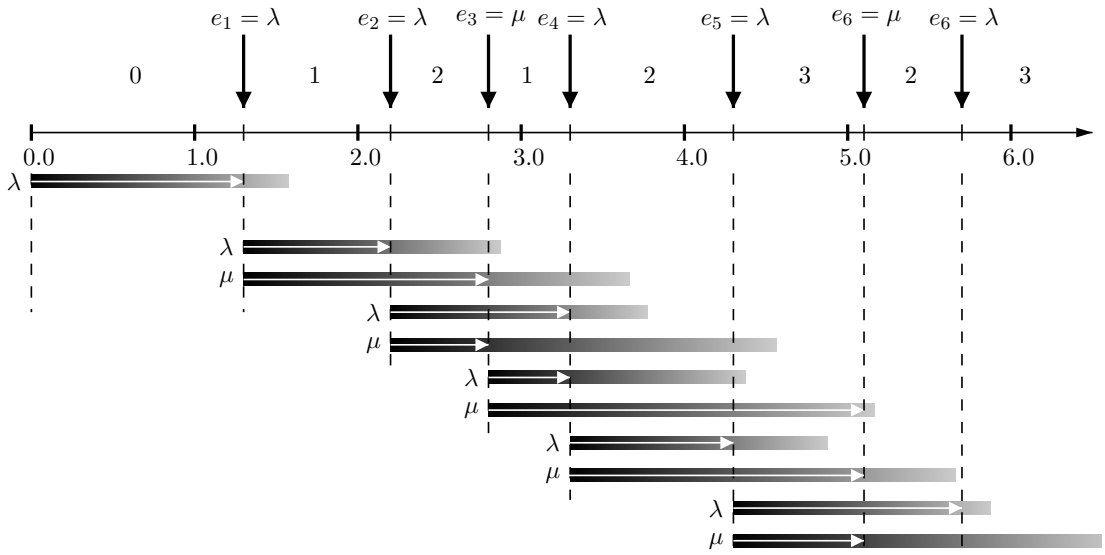


Figure 2.6: The stochastic timeline of the M/M/1/n stack is depicted.

However, there is a big difference, that is each simulation has an uncertainty in the output. Now, the events are not governed by the simple structure of static lifetimes but guided by stochastic clocks that generate, according to this, the samples that is one lifetime (as we can see the shadowing bars).

We can see two type of events, λ and μ, which can be marked as active event or inactive depending on the present state. Thus, an active event can be an *old clock* or a *new clock*, and conversely an inactive event has an *inactive clock*. An *old clock* is achieved by the subtraction of the clock values of the triggered events until this old clock is triggered, i.e., it is assigned a original value minus every clock values that are triggered until then.

The definition of the inactive clocks, the new clocks and the old clocks is given by

$$\begin{aligned} C(e_n, i) &= -1,\ e_n \notin E(S_i) \\ C(e_n, i) &= \mathcal{F}(\cdot, S_i, S_{i-1}, e_n),\ e_n \in N(S_i, S_{i-1}, e) \\ C(e_n, i-1) - C(e^*_{i-1}, i-1) &= C(e_n, i),\ e_n \in O(S_i, S_{i-1}, e) \end{aligned} \tag{2.18}$$

where n is the number of events, i is the i^{th} steps of simulation, and $\mathcal{F}$ is a probabilistic distribution function. Therefore, the $E(s)$ is a function of a set of active events in a state s, and $N(s, s', e)$ is the function of *new clock* in respect to an event and a transition to

the s' state to s. Consequently, we define the $e_n \in O(S_i, S_{i-1}, e)$ based on the premise $e_n \notin N(S_i, S_{i-1}, e)$ and $e_n \in E(S_i)$.

In this chapter we have reviewed the basic principles of DES and SDES as well as the specification of stochastic processes. However, an algorithm that estimates the scheduling of GSMP is needed in order to solve the learning methodology in a complete way. In the next chapter we shall review the state of the art concerning learning stochastic systems and this will serve as foundation for the specification of a new learning algorithm for GSMP and its abstraction for continuous systems.

Chapter 3

Related work

In the previous chapter we have given the main definitions and the essential background necessary to understand the concept of timed automaton, stochastic automaton, DES and SDES. In this chapter we review the state of the art of stochastic model learning, . In particular we shall concentrate on: learning generalized semi-Markov chain (see section 3.1), its abstractions for continuous systems including other approaches (see section 3.2), and two approaches that can verify and test stochastic models (see section 3.3 and 3.4). Not that this related work is centered around the development of the basic ideas of learning, verifying, and testing.

We begin with reviewing the learning algorithms for a set of stochastic models like probabilistic/stochastic automata (that identify a stochastic language), the discrete-time and continuous-time Markov chains, and briefly the hidden Markov chains. Next, a few other approaches to abstract the continuous systems are discussed. This is followed by recent work about statistical model checking, on which tools have been developed that allow the verification of the learned models. Lastly, we describe the existent testing methods that are able to test stochastic models.

However, since the models (automata, Markov chains, etc) are directly related with DES or SDES we could use probabilistic/statistical model-checkers to check them. Moreover we can generate a set of tests from these models, unit tests or stochastic tests. Also, the learning approach can be used to learn models, which can then be verified and tested.

3.1 Learning stochastic and probabilistic models

Learning algorithms are widely used for system analysis and system modeling. As we have said in the previous chapter, the verification and test of stochastic systems is our main goal. Machine learning solved several problems but in most cases does not ensure

its reliability. So, today some models have scarce reliability. Here we discuss some probabilistic/stochastic learning methods where our goal is to use statistical model checking to verify it.

The stochastic models are used for reliability and performance analysis of a set of complex systems. As we have presented in the previous chapter various models are available. Now, we describe some related algorithms for its learning. Carrasco and Oncina [1994] introduced the criterion of learning stochastic regular grammars. They proposed an algorithm based on state merging to learn a probabilistic automaton. This method begins by the construction of one prefix tree from a set of output sequences provided by one implementation (called *sample executions*). Next, they established a well defined stable relation (the *state equivalence*) to merge equal states. This process produces a stochastic automaton that recognize a stochastic language. It is based on the beginning principle of language identification in the limit which was introduced by Gold [1967]. He also proved that regular languages cannot be identified if only text is given, but they can be identified if a complete presentation is provided (i.e. "Is the information sufficient to determine which of the possible languages is the unknown language?"). Later, Carrasco and Oncina [1999] propose the same solution but in polynomial time.

A more close method is the work developed by Kermorvant and Dupont [2002]. They present a new statistical framework for stochastic grammatical inference algorithms based on a state merging strategy. They use the multinomial tests for establishing the equivalence relation between the states. Their approach has three advantages. First, the method is not based on asymptotic results, and thus small sample case can be specifically dealt with. Second, all the probabilities associated to a state are included in a single test (*chi-square test* - $\mathcal{X}^2$). Third, a statistical score is associated to each possible merging operation and can be used for best-first strategy.

Given the good results of the learning probabilistic automatons, Sen et al. [2004b] propose a new extension to learn the continuous-time Markov chains. Their learning algorithm consists in learning a model from an edge labeled continuous-time Markov chain (see section 3.3). Moreover, with this method we can use the learned model from a set of practical systems (e.g, the industrial systems, the avionic systems, and the automobile systems) as the input model for a set of available tools. These tools allow the analysis, verification and testing of Markov chains.

Another closely related work is proposed by Wei et al. [2002]. They propose a method to learn continuous-time hidden Markov chains and also propose the acquisition process with fixed length of sample executions (the sample executions have a finite and a static length defined before all the learning process). This causes a serious trouble in the learning

process if the specified size is not large enough to learn the model, which is known as the problem of insufficient training data. The learning theory shows that the values computed by the training algorithms converge to the correct probabilities if the amount of training data tends to infinite. In practice an existing bias is observed. Which one depends on the number of training samples and on the length of the samples. When the length increases, the likelihood of a given sequence decreases.

In the following, we explain in detail the learning approaches which will form the basis of the next chapter.

3.1.1 Learning continuous-time Markov chains

We explain here some details of the learning methodology proposed by Sen et al. [2004b]. They proposed an algorithm also based on state merging paradigm introduced by Carrasco and Oncina [1994]. The *prefix tree* is constructed given a set of sample executions. A prefix tree has two fields coupled to each node, which are the following: expectation value to each transition branch, and clock samples to each label in a prefix tree node. For example, figure 3.1 illustrates a prefix tree with five nodes generated by three paths. It has the transition labels $\{a, b\}$, P that is the expected value to the respective branch, and C the clock samples from transitions.

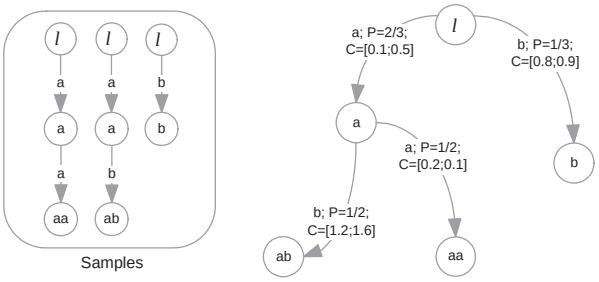


Figure 3.1: Example of a prefix tree constructed from three sample executions. It has five nodes and four transitions each one annotated.

A prefix tree is used to find similar states which is the essence of this state merge paradigm. When two similar states are found, they should be merged to give rise to a new model. This process is made recursively for each node in the prefix tree. After that process one generates a Markov chain, which in this case is an edge labeled Markov chain. As we have referred, this is an extension of a known continuous-time Markov chain. This chain has one label (an identification symbol) coupled to each transition and it solves the nonexistence of transition symbol in CTMC.

The proof of concept in this case is an indispensable task. Thus, Sen et al. [2004b] demonstrate that given a structurally complete sample to his learning algorithm, in the

limit (as sample executions grows to infinite), the learned model is similar or equal to the original model.

There is other approach that we will not explain here that is based on empirical rules to learn and classify Markov chains [Lowd and Davis, 2010].

3.1.2 Learning hidden Markov chains

The hidden Markov chains [Barbu and Limnios, 2008] have become increasingly popular in the last several years, for different reasons according to Rabiner [1990]. There are two strong reasons why this has occurred. First the models are very rich in mathematical structure and hence can form the theoretical basis for use in a wide range of applications. Second the models, when applied properly, work very well in practice for several important applications.

We pointing here some related work about hidden Markov chains in order to know how to learn the *observable process*.[1] Rabiner [1990] proposes a method to learn hidden Markov models in order to provide a method for voice recognition. Wei et al. [2002] also proposes a method to learn continuous-time hidden Markov chains for modeling network performance.

However, our objective here is to make it known that there are other types of models (with a hidden part) that could be verified or tested.

3.1.3 Learning stochastic languages with artificial neural networks

The artificial neural network (ANN) introduced by McCulloch and Pitts [1988] describes a formal approach of the human brain. The ANN aims to emulate the human thinking process, and it is constructed by a binary network of neurons in which each neuron is composed by one activation function and some synopses. The synopses are the input signals for the activation functions of neuron, which are provided by other connected neurons in the ANN.

Carrasco et al. [1996] have introduced the criterion of learning stochastic grammars, which are mentioned in the previous chapter, but also another learning method based on ANN. In his paper he affirms that the generalization ability of their method is acceptable, and that the second-order recurrent neural network may become a suitable candidate for modeling stochastic processes. However, he does not propose an analysis of his approach, and does not guarantee some important properties of the ANN method, which are ensured in the method based on MCs.

[1] The hidden Markov chains have one observable and one hidden process.

The ANNs models have advantages and disadvantages over the MCs. The ANNs have a variety of learning methods, which allows to obtain easily a reasonable model. However, in many cases the reliability of this model is unobtainable, and an analysis of an ANN is intractable, due to its complexity and non linearity. The stochastic models are clearly better in analysis and verification. This is due mainly to the developed statistical/probabilistic model checking tools (see 3.3).

The stochastic solution is more expensive to calculate, nevertheless verifying properties in learned models is more efficient. Considering the advantages and disadvantages of the various approaches stochastic models are overall the best option.

3.2 Discrete event systems specification

The *discrete event system specification* (called DEVS) is a specification that it is commonly used in industry. DEVS is a modular and hierarchical formalisms for modeling and analyzing general systems (Wainer [2009] has written a very interesting book about DEVS). This systems are described by: discrete state systems, continuous state system that can be described by differential equations, and hybrid state systems that it is continuous and discrete state spaces. There is a Matlab toolbox, SimEvents, that includes DEVS.

There are some developments about discrete simulation of continuous systems as proposed by Nutaro [2005, 2003]. He proposes a parallel algorithm for DEVS in order to simulate approximations of continuous systems that are provided by the quantization of *ordinary differential equations* (ODE). However, the DES are characterized by asynchronous and irregular or random executions. So, finding a parallel algorithm is a challenge. As is well known, DEVS is the closer specification of the foundations of discrete approximations of continuous systems. Therefore, this specification should be seen in a different way from the DES approach exemplified in the previous chapter. An another contribution that uses the same approach, called *quantized state system solver* (QSS), is by Cellier et al. [2007]. It uses two algorithms to quantize the discrete state space of an ODE and produce an equivalent DEVS. This replaces the classic time slicing by a quantization of the states, leading to an asynchronous discrete-event simulation model instead of a discrete time difference equation model. Also discussed in that paper are the main properties of the methods in the context of simulating discontinuous systems (the asynchronous nature of these algorithms gives them important advantages for discontinuity handling).

An example of the use of DEVS to make an abstraction of continuous systems is proposed by Carmona and Giambiasi [2007]. They use the DEVS with an extension called *generalized discrete event modeling* (G-DEVS) in order to make the discretization of the

state space for an integrator with linear and polynomial segments. Moreover, in case of input discontinuities its remarkable behavior, strongly contrast with the poor solution of classical numerical solvers. More recently Castro et al. [2009] proposed a formal framework for stochastic DEVS, including their modeling and simulation.

We describe in chapter 4 the DEVS in a more detailed manner.

3.3 Probabilistic/Statistical model checking

The goal of model checking technique is to try to predict system behavior, or more specifically, to formally prove that all possible executions of the system conform to the requirements [Baier and Katoen, 2008]. Thus, probabilistic model checking focuses on proving correctness of stochastic systems (i.e., systems where probabilities play a role).

However, the quantitative analysis of stochastic systems is usually made using the reward variables, but in many cases this is not enough to validate some requirements. So, quantitative properties of stochastic systems usually specified in logics are used to compare the measure of executions that satisfies certain temporal properties with thresholds. The model checking problem for stochastic systems with respect to such logics is typically solved by a numerical approach, Kwiatkowska et al. [2011, 2008], that interactively computes the exact measure of paths satisfying relevant sub-formulas. Another approach to solve the model checking problem is to simulate the system for finitely many runs, and use hypothesis testing to infer whether the samples provide a statistical evidence for the satisfaction or violation of the specification (called *statistical model checking*). A recent overview of statistical model checking is presented by Legay et al. [2010].

At the moment of writing of this thesis there is many probabilistic/statistical model checker tools such as: UPPAAL, Prism, MRMC, Vesta and Ymer. Some of them such as UPPAAL (i.e., their new extension of statistical model checking) and Prism (verification of the real-time probabilistic systems) are clearly two mature tools [David et al., 2011, Kwiatkowska et al., 2008].

Oldenkamp [2007] has made a comparison between known probabilistic model checkers. They described that Ymer is more accurate than Vesta [Younes, 2004, Sen et al., 2005] and they made several justifications for that. The statistical model checking formalism was invented and introduced by Younes et al. [2010]. They also have introduced the verification of black-box systems but with some restrictions on verifiable unbounded properties [Younes, 2005, Sen et al., 2004a].

Younes [2004] proposes a unified logic and a statistical method to verify steady state properties. Rabih et al. [2011] also proposes other method for the verification of steady

state properties[2] for very large systems. Other approach based on Bayes theorem has emerged to check other systems like biological systems [Jha et al., 2009].

Statistical abstractions and model checking of large heterogeneous systems was also proposed by Basu et al. [2010]. An heterogeneous system is a series of interconnected parts (computer systems) that act together in a common purpose or produce results impossible by action of one alone. Their paper proposes the creation of a stochastic abstraction manually for their application. So, this smaller model can be verified using efficient techniques such as *statistical model checking*. They have applied their techniques to an industrial case study, the cabin communication system of an airplane (a model to synchronize all on-board systems/computers).

3.4 Statistical model-base testing generation

Software development for discrete and continuous systems is an error-prone task. Moreover, several projects use unit test generation to validate their coverage. Testing programs can be used to make several shots in the program domain (i.e., only detect the presence of bugs in an execution). But, program testing is incomplete and it does not cover overall program domain but only a finite part. So, we have to use verification techniques like model checking to do program verification. To be sure that the code solves the right problem, we must have a specification that describes what we want the program to do. So, describing it in a formal model that checks some requirements (properties), avoids many troubles such as the need for the code to be reworked or discarded. We describe here some related work about model-based testing, which allows automatic generation of tests for implementations based on models.

In Fraser et al. [2009] an overview of model-base testing approaches is presented. They are techniques to test programs using unit tests generated from formal models that can be used in classical model checkers. At the moment of the writing of this thesis there is no any approach to testing with probabilistic/stochastic model checkers. Initially this was due to a lack of statistical model checkers generating counter example paths. However, recently some papers have emerged that propose mechanisms to the generation of counter-examples [Aljazzar et al., 2011, 2010, Han et al., 2009].

We describe some results about testing of stochastic systems and also about mutation testing. Merayo et al. [2009] proposes a formal framework to test systems where nondeterministic decisions are probability quantified and temporal information is defined by

[2]Properties that are unchanging in time. The probabilities that various states will be repeated will remain constant.

using random variables. They propose in their paper an extension of the classical finite state machines formalism in order to define the stochastic systems. Thus, they have introduced the notion of conformance relation (establishing what a good implementation is) and of test case (describing what are stochastic tests). Other close work about statistical tests is proposed by [Ševčíková et al., 2006]. Hierons and Merayo [2009] propose the concept of mutation testing for probabilistic and stochastic finite state machines.

The related work of learning algorithms for probabilistic and stochastic models was presented here. We also described the abstractions for continuous systems, the statistical model checking techniques, and the statistical test generation for stochastic models. In the following chapters, we describe our contributions for the state of the art of learning stochastic processes.

Chapter 4

Learning and testing stochastic models

In this chapter we present the core foundations of the contribution of this thesis. The mathematical theorems and definitions introduced in the previous chapters will be used as a basis here. Furthermore the extensions of the previous definitions will be introduced when needed.

In section 4.1 we extend the definitions for our learning process and in section 4.2 we present the learning approach for GSMP based on these new extensions. We present, in section 4.3, the correction of the learning approach. We propose also, in section 4.4, a method to model the perturbations for continuous systems as well as explain our efforts to model a subclass of these systems in order to check some requirements. Given a realistic model achieved by the application of our learning process, we will show, in section 4.5, how a test oracle can be achieved for deterministic and stochastic models. Lastly, in section 4.6, we give an overview of SDES toolbox, our contribution for Matlab.

4.1 Preliminary definitions

In this section we begin by introducing the definitions required for the learning process, and also we define concepts such as paths, prefix tree, and probability measure of path. Moreover, we establish a well defined equivalence relation in order to define when states are equivalent/similar. Lastly, we describe a solution for the problem of non deterministic merging.

A *infinite path* of a $GSMP = (\mathcal{X}, \mathcal{E}, \Gamma, p, p_0, G)$ is a sequence $\rho = a_0 \xrightarrow{e_1,t_1} a_1 \xrightarrow{e_2,t_2} a_2 \xrightarrow{e_3,t_3} \cdots$ where each a_i is a state, $e_i \in \mathcal{E}$ is an occurred event at time $\sum_{x=1}^{i} t_x$, and $t_i \in \mathbb{R}_{\geq 0}$ is the holding time of each event, for all i in $\mathbb{N}$. One says that $s = a_0$ is the initial

state of the sequence.

A *finite path* of a GSMP is defined by $\pi = s_0 \xrightarrow{e_1,t_1} s_1 \xrightarrow{e_2,t_2} s_2 \xrightarrow{e_3,t_3} \cdots \xrightarrow{e_n,t_n} s_n$, where $s_0 = s$ is the initial state, $s_i \in \mathcal{X}$ is the state at i^{th} step, t_i is defined as in previous definition, and n is the length of path, where $0 < i \leq n$, for all i in $\mathbb{N}$.

We need to establish that a set of *finite paths* acquired by a GSMP are equivalent to a set of finite paths accepted by the prefix tree. From the definition 8 we know that the defined prefix tree (see below) is a particular case of a GSMP. Moreover, the prefix tree constructed from a set of paths is always a particular case of GSMP, even without any equivalence between the states. However, we cannot ensure here that the model that was simulated is equal or equivalent to the model identified by the prefix tree (as we shall see later, the *learning in the limit*).

Remark 4.1. Note that the prefix tree with two equivalent states could establish a loop transition, which is not allowed in a prefix tree. Thus, we need to establish the inclusion with a stochastic automaton in order to always ensure that the converted prefix tree in this case continues to be a GSMP. The prefix tree is a particular case of a stochastic automaton, but the reverse it is not true.

A *prefix tree* that has an acceptor S, a set of *finite paths* (n-samples, denoted by S_n), is a tree $Pr(S) = (\mathcal{Q}, F, \delta)$, where $\mathcal{Q}$ is the set of the sequence of events accepted by S, F is a set of entire words (i.e., a sequence of events) accepted by S ($F = S$), and δ is the transition function which have the following definition,

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

A sequence of events $e_1\, e_2\, e_3\, \ldots\, e_n$ defined by the prefix tree that accepts $\pi = s_0 \xrightarrow{e_1,t_1} s_1 \xrightarrow{e_2,t_2} s_2 \xrightarrow{e_3,t_3} \cdots \xrightarrow{e_n,t_n} s_n$ is denoted by $\pi|_{\mathcal{E}}$. For a given path π starting at s state where $s = s_0$, we extend our definitions in order to simplify some notation in the above descriptions and algorithms, as follows:

- $\pi|_{\mathcal{E}}[s, i]$ is the i^{th} event of the event sequence that begins in state s,
- $\pi|_{\mathcal{X}}[s, i]$ is the i^{th} state of the state sequence that begins in state s,
- $\pi|_{G}[s, i]$ is the i^{th} holding time of the event sequence ($\pi|_{\mathcal{E}}[s, i]$) that begin in s state,

- $\eta(\pi|_{\mathcal{E}}[s,i]) = \pi|_{\mathcal{X}}[s,i-1]$ is a function that returns the state associated to an event e_i,
- $\varepsilon(\pi|_{\mathcal{X}}[s,i]) = \pi|_{\mathcal{E}}[s,i+1]$ is a function that given a state of a path returns its associated event,
- $\delta(\pi|_{\mathcal{E}}[s,i]) = \pi|_{G}[s,i]$ is a function that given an event $\pi|_{\mathcal{E}}[s,i]$ returns its holding time $\pi|_{G}[s,i]$,
- $\tau(s,xe_i)$ is a function that gives the set of next 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 $x\,e_i$, for instance from $\{x\,e_i\,e_j, x\,e_i\,e_k, ...\}$ we get $\{e_j, e_k, ...\}$,
- a map function $\sigma(\pi|_{\mathcal{X}}[s,i]) = u$, where $u \in \mathcal{Q}$ is a sequence of events accepted by the prefix tree $Pr(\pi|_{\mathcal{E}})$, and
- $\varrho(s,xe_i)$ is a function that gives the holding times associated at each word xe_i in a prefix tree $Pr(\pi|_{\mathcal{E}})$.

Now, we will need to ensure that our prefix tree is and will remain a generalized semi-Markov process. For this reason, we propose definition 8 as the definition of inclusion that $Pr(S)$ is a GSMP, or in other words a stochastic automaton.

Definition 8. The prefix tree $Pr(S) = (\mathcal{Q}, F, \delta)$ for a set of multiple sequences S is a particular stochastic automaton, i.e., $PSA(S) = (\mathcal{X}, \mathcal{E}, \Gamma, p, p_0, G)$, where

1. $\mathcal{X}$ = accepted sequences from $\mathcal{Q}$,
2. $\mathcal{E}$ = unique events from $\mathcal{Q}$,
3. $\Gamma(s_i) = \tau(s, \sigma(s_i))$,
4. $p(s',s,e) = \begin{cases} 1 & \text{if } \delta(\sigma(s),e) \neq \perp \text{ and } \sigma(s') \neq \perp \\ 0 & \text{otherwise} \end{cases}$,
5. $p_0(s) = 1$, and
6. G are the estimated probabilistic distributions of a set of multiple path samples.

A PSA is a GSMP consistent with the sample in S. For all n-samples S_n there exists a corresponding path in the GSMP. Following the definitions of correspondence between GSMP and a prefix tree, we will define the equivalence relation between two states. This relation creates a more abstract GSMP from a given set of sample executions, where the size of model is reduced on each equivalence between states.

We introduce a well defined stable relation (definition 9) in order to establish the correct equivalence of states.

Definition 9. Let $\mathcal{M} = (\mathcal{X}, \mathcal{E}, \Gamma, \mathbf{p}, p_0, \mathbf{G})$ be a stochastic automaton, a relation $R \subseteq \mathcal{X} \times \mathcal{X}$ is said a *stable relation* if and only if any s, s' have the following properties,

$$|\Gamma(s)| = \left|\Gamma(s')\right| \tag{4.1}$$

there is a one to one correspondence f between $\Gamma(s)$ and $\Gamma(s')$ such that for any event $e \in \mathcal{E}$,

$$\begin{aligned} &\text{if } \exists\, n \in \mathcal{X} \text{ such that } \mathbf{p}(n, s, e) \neq 0, \text{ then} \\ &\exists\, n' \in \mathcal{X} \text{ such that } \mathbf{p}(n', s', f(e)) \neq 0, \\ &\mathbf{G}(s, e) = \mathbf{G}(s', f(e)), \text{ and } (n, n') \in R \end{aligned} \tag{4.2}$$

and conversely,

$$\begin{aligned} &\text{if } \exists\, n' \in \mathcal{X} \text{ such that } \mathbf{p}(n', s, e) \neq 0, \text{ then} \\ &\exists\, n \in \mathcal{X} \text{ such that } \mathbf{p}(n, s', f(e)) \neq 0, \\ &\mathbf{G}(s', e) = \mathbf{G}(s, f^{-1}(e)), \text{ and } (n', n) \in R \end{aligned} \tag{4.3}$$

where $|\Gamma(s)|$ is the number of active events in the state s, $\mathbf{p}$ is a probabilistic transition function, and $\mathbf{G}$ 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 $(s, s') \in R$.

Example 4.2. As a concrete example, given $|\Gamma(s)| = |\Gamma(s')| = 2$, $\Gamma(s) = \{a, b\}$, $\Gamma(s') = \{c, d\}$, satisfies equation 4.1 (i.e., the set of active events for s and s' has the same size), and also satisfies $\mathbf{G}(s, e) = \mathbf{G}(s', f(e))$ if $\mathbf{G}(s, a) = \mathbf{G}(s', c)$ and $\mathbf{G}(s, b) = \mathbf{G}(s', d)$, or $\mathbf{G}(s, a) = \mathbf{G}(s', d)$ and $\mathbf{G}(s, b) = \mathbf{G}(s', c)$. From equation 4.2 and equation 4.3 we know that a composition between all states reachable by s and all states reachable by s' must be also a stable relation.

Remark 4.3. The proposed definition 4.1, 4.2 and 4.3 do not need to know the event identifiers and its probability distribution.

The correctness of a learning algorithm crucially depends on the fact that merging two equivalent states results in a GSMP that generates the same model. Thus, the merge of two states due to the existence of equal active event sets creates a non deterministic choice, which needs to be solved. We propose two solutions for two distinct situations. First, if it is known a priori that the model to learn is a GSMP we apply a deterministic merge from equal nodes, i.e., we know that a subsequent evolution of two equal states will lead to similar states. We need to merge them recursively while there exists two equal states s' and s'', i.e., $While\ (\exists\, s, x \in \mathcal{Q}, e \in E : s', s'' \in \sigma(s, x\, e))\ merge(s', s'')$.

Example 4.4. Let two non-deterministic transitions labeled with same event e after merging s_2 in s_1. We initially have $\rho(s, x\,\sigma(s_1)) = e$ and $\rho(s, x\,\sigma(s_2)) = e$ that are equal sets, the merge of this two states is only possible if we merge states s_3 and s_4 given inversely by $\varepsilon(s_3) = e$ and $\varepsilon(s_4) = e$. So, merging s_3 and s_4 may create more non deterministic transitions. This process can be repeated recursively until there are no another non-deterministic transition.

Second, it is not known a priori that the model to learn is a GSMP, in this case we need to know probabilistically if the events are equal or not. We do not need to know the name of events but only the empirical distribution for each one. We will describe this in a more detailed manner in the section 4.4.

4.2 Learning generalized semi-Markov processes

The proposed learning methodology is based on the state merge technique over a prefix tree constructed from sample executions. In this case we refer to learning processes that have a state age memory, which does not occur in processes with Markov property. Thus, we need to estimate the history of clock states, which can be labeled as *old clock*, *new clock* or *inactive clock*. Furthermore an estimator is needed in order to predict the distributions and parameters from sample data. The partial correctness of the learning method is also described below as learning in the limit. For the method that we will describe, we guarantee that when the sample grows, the probability error of merging two non-equivalent states, in the limit, goes to zero.

We propose the algorithm 1 for estimation of past states of the clocks; the algorithm 2 that allows testing the similarity between two states and also construct the stochastic timed automaton; and lastly the algorithm 5 that estimate the parameters for probability distributions.

4.2.1 Scheduling as state age memory

As we have seen in chapter 2, the Markov processes are characterized by the two properties 2.6 and 2.7. However, the GSMP relaxes the property 2.7 by allowing different distributions to each inter-event time (i.e., in GSMP the inter-event times are not equal, unlike in a CTMP that is governed by a exponential distribution). In general, the GSMP allows the use of probability distributions like Weibull, Log-normal and Normal (without memoryless property) for specifying holding time of states.

Due to the relaxed property, data structures like a Fibonacci heap are needed in order to simulate a GSMP. A Fibonacci heap memorizes the current state of clocks and refresh

Algorithm 1: Scheduler estimator (SE)

input : A set of paths S of size $|S|$ and a prefix tree $Pr\,(S)$.
output: A prefix tree $Pr\,(S)$ with modified clock samples.

```
for n ← 1 to |S| do                                              // For all paths n
    for l ← 2 to |S_n| do                                        // For all nodes l of path n
        for p ← l to 1 do                                        // Decrement p
            if ¬(ε(S_{n,l}) ∈ τ(σ(S_{n,p})) ∧ |τ(σ(S_{n,p}))| > 1 ∧ ε(S_{n,p}) ≠ ε(S_{n,l})) then
            p ← p + 1; break;
        if S_{n,p} ≠ S_{n,l} then
            Val ← 0;
            for t ← p to l do                                    // Estimate the original clock value
                Val ← Val + δ(ε(S_{n,t}));
                if S_{n,t} = S_{n,l} then break;
            replace_clk(Pr(S), σ(S_{n,l}), Val) ;                // Replace the estimated clock value
```

its clocks. In SDES the events compete each other to trigger the winner event which have the smaller time. We proposed a schedule estimator (SE) in order to estimate the past clock states and infer the original clock value.

Example 4.5. Suppose two events a and b in a state s, two random variables $X_a \sim E(0.2)$ and $X_b \sim W(1, 0.1)$. Assuming the state s as initial state, the events a and b are labeled as *new clock*. When this initial state occurs a sample from random variable is achieved, in this example $x_a = 1.2$ a sample value of X_a and $x_b = 0.5$ a sample of X_b. Then event a and b compete each other to trigger the event with the smaller value. In this case, the winner is the event b. Furthermore the sample of event a is saved in a scheduler to be used in the next time and labeled as *old clock*.

The purpose of the algorithm 1 is to estimate when the value of a clock is labeled as *new clock* or a *old clock*. For the learning algorithm only the *new clock* values are suitable to predict the probabilistic distributions of each event.

Remark 4.6. Note that in the algorithm 1, S is characterized as follows: S_n is the n^{th} path of S and $S_{n,l}$ is the l^{th} element of path n, for each $0 < n \leq |S|$ and $0 < l \leq |S_n|$.

Our *scheduler estimator* algorithm is proposed in order to solve the estimation of original values from clock distributions. This estimation happens due to the existence of a mapping function between sample executions S and the prefix tree which include the same samples $Pr(S)$. We define the map function as one relation between sample execution nodes and prefix tree nodes, which allow to know the produced node in the prefix tree by the sample execution node. So, we have a form to traverse each sample

execution and compare child sets of the prefix tree in order to predict the label of each clock in a sample execution.

We explain in the following how algorithm 1 estimates original sample clock values. First, the algorithm begins by traversing each path of the 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, in one sample execution, until we have encountered a possible *inactive clock*. When an *inactive clock* is encountered for the current event this implies that this event is not in its active event set (given by σ function). Therefore in the worst case the first element of the sample execution can be encountered. Given this element we can reconstruct the original clock value by the sum of the values between the found element and the current state. Lastly, we replace the *old clock* value by the estimated original clock value.

Example 4.7. Considering the model of figure 4.1, a set of samples from the M/M/1 stack of the figure 2.5, illustrates the prefix tree $Pr(\{s_1, s_2, s_3, s_4\})$ produced by four *sample executions*[1] $\{s_1, s_2, s_3, s_4\}$. In the following we exemplify the prediction of the original clock value μ_1. First, the model initializes at state 0 and only the event λ can be activated. In this case it is triggered with holding time of $0.86s$. Second, we can choose between event λ and μ since it is in state 1. Suppose that a clock value for each event is acquired, respectively $0.79s$ and $0.90s$. Now, the winning event is λ with value $0.79s$ and μ is scheduling to next time with value $0.90 - 0.79 = 0.11s$. Next, to following the same method is acquired the value $0.20s$ for event λ. Certainly μ wins with value 0.11 but with a different value from the first value acquired. In a descendant order of the simulation trace, we reconstructed the real value of μ through a sum of values from sample execution until the event μ labeled as *inactive clock* is found. The original clock value of μ is $0.11 + 0.79 = 0.90s$.

The figure 4.1 illustrates the behavior of a SDES scheduler. The black trace exemplifies how we can estimate original samples of clocks knowing this prefix tree and the path (set of black arrows). The sample clock values for each state are indicated inside brackets where $t_{\mathcal{F}_\lambda}$ is the sample value from a random variable which follows a probabilistic distribution of the λ event, and $t_{\mathcal{F}_\mu}$ denote the same but for the μ event.

[1] Simulation outputs of a stochastic process.

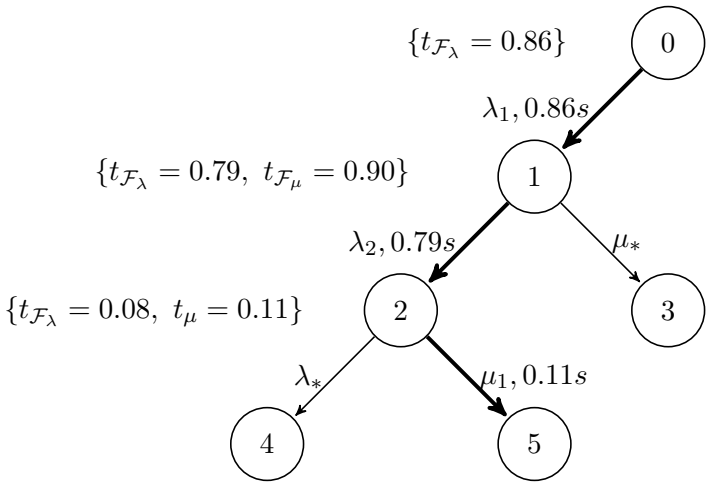


Figure 4.1: The example of discrete event system scheduling. This set of transitions and states forms a sample of one prefix tree with annotations.

4.2.2 Testing similarity of states

We propose algorithm 2 in order to test the similarity between states, merge them, and construct a deterministic stochastic timed automaton. This algorithm is created in order to support an extension of Markov processes, the generalized semi-Markov processes. Our approach is different from the other approaches proposed by Carrasco and Oncina [1994] (learn stochastic languages) and Sen et al. [2004b] (learn continuous-time Markov chains).

Remark 4.8. Note that in algorithm 2 we denote C^2 as follows: C_c is the c^{th} cluster of C and $C_{c,n}$ is the n^{th} element of cluster C, for each $0 < c \leq |C|$ and $0 < n \leq |C_c|$. The `is_active` and `inactivate` functions allow that only the prefix tree words that were not merged are used. The `inclusion` function converts a prefix tree into a stochastic timed automaton.

Our *probabilistic similarity of states* algorithm is subdivided in three blocks. The first block is composed by a `clusterize` function that clustering the states with an equal active event set (given by τ function). We achieve with `clusterize` function a plain static equivalence between states, nevertheless we need to establish a while cycle with `count` $>$ `0` to cover the other cases when `deterministic_merge` changes clock samples of the similar states. With this `clusterize` function we guarantee the equation 4.1, which says that only states with event sets of the same size can be merged. A performance comparison between our algorithm and the others cited in the state of the art [Carrasco and Oncina, 1999, Kermorvant and Dupont, 2002] was made. We conclude that their computational structure are quite similar, but considering performance our method is

[2] A set of clusters classified by events.

Algorithm 2: Probabilistic similarity of states (PSS)

input : A prefix tree $Pr(S)$, and a type I error α.
output: A stochastic timed automaton $\mathcal{M} = (\mathcal{X}, \mathcal{E}, \Gamma, p, p_0, G)$.

$\mathcal{M} =$ `inclusion`$(Pr(S))$; `// By the inclusion of definition 8 between the prefix tree and the stochastic timed automaton.`
count $\leftarrow 1$;
while count > 0 **do**
  count $\leftarrow 0$;
  $C \leftarrow$ `clusterize`$(\mathcal{M})$; `// Clustering by active event set` $\tau(s')$ `for each node` s' `of the stochastic timed automaton` $\mathcal{M}$ `that initially is equivalent to` $Pr(S)$.
  for $c \leftarrow 1$ **to** $|C|$ **do**
    for $n \leftarrow 1$ **to** $|C_c|$ **do**
      $x \leftarrow n + 1$;
      while $C_{c,x} \neq C_{c,|C_c|}$ **do**
        if `is_active`$(C_{c,x})$ **then**
          if `similar`$(C_{c,n}, C_{c,x}, \alpha)$ **then** // $\tau(C_{c,n})$ `and` $\tau(C_{c,x})$ `sets are similar`
            `deterministic_merge`$(\mathcal{M}, C_{c,n}, C_{c,x}, empty\_H, empty\_PT)$;
            `inactivate`$(C_{c,x})$;
            count $\leftarrow$ count $+ 1$;
        $x \leftarrow x + 1$;

fastest due to a selection method based on this clustering methodology.[3] In the second block we use the `similar` function to test when two states are similar. This function is defined in algorithm 3 and it uses the Kolmogorov-Smirnov test (A.2.3) to decide if two empirical probabilistic distributions are equal. It verifies whether there exists a one to one correspondence of events between two active event sets through a statistical equivalence. If there is a correspondence for all events of an active event set, the equation 4.2 is satisfied. Lastly, the algorithm 2 merges the equal states by the function `deterministic_merge`. It initializes the construction of the stochastic timed automaton. This function defined in algorithm 4 solves the problem of non-deterministic merge of states when two states have the same set of events.

The algorithm 3 is used to test the similarity between two active event sets E_1 and E_2 within the type I error α, as described previously. The Kolmogorov-Smirnov test (known as goodness of fit test) is applied to compare two empirical distribution functions with hypothesis H_0 : the distributions are equal, against H_1 : the distributions are different. An α is the error of rejecting a true null hypothesis (H_0) in an hypothesis test (see A.2). The function applies the hypothesis test, knowing the empirical CDF F_{n_1} and F_{n_2}, for

[3]Our method can be implemented as a parallel algorithm to increase the performance and to support more complex systems.

Algorithm 3: `Similar` function

input : Two sequence of events s_1 and s_2, and α a type I error
output: A boolean, True if similar otherwise False

$E_1 \leftarrow \tau(s_1)$; $E_2 \leftarrow \tau(s_2)$;
`assert`$(|E_1| \geq |E_2|)$;
foreach e_1 *in* E_1 **do** `// Comparing two active event sets` E_1 `and` E_2
  while $E_2 \neq$ **do**
    $e_2 \leftarrow$ `get_element`(E_2);
    $F_{n_1} = \mathcal{T}(\varrho(s_1 e_1))$; `// Constructing empirical distribution of clock sample set;`
    $F_{n_2} = \mathcal{T}(\varrho(s_2 e_2))$; `// Constructing empirical distribution of clock sample set;`
    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 `similar`$(\delta(s_1 e_1), \delta(s_2 e_2), \alpha) \neq True$ **then**
        return False;
      break;
    `put_element`(E_2, e_2);
if $|E_2| < 1$ **then** return True; **else** return False;

two clock samples sets with size n_1 and n_2 respectively. We denote as $\mathcal{T}$ a function for constructing the empirical cumulative distribution from a set of sample clocks, i.e.,

$$\mathcal{T}_n(x) = \frac{number\ of\ z_1, z_2, ..., z_n\ that\ are \leq x}{N} \tag{4.4}$$

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. As seen in the preliminary definitions, the function ϱ returns the collected sample clock data for one event from the prefix tree.

Ensuring a merge without non-deterministic choices for which the algorithm 2 is required. Thus, we propose a recursive algorithm 4, a variant was defined previously in section 4.1, which needs to know a priori that the sample executions are provided by a GSMP. The recursion is defined with three base cases, as follows: two states s_1 and s_2 are equals, the state s_2 does not have any active event, and the state s_1 or state s_2 is merging 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 algorithm of the *deterministic merge* function begins by a comparison between state degrees in order to merge the state with higher degree with the state with lower degree. Next, every state that is removed from the deterministic stochastic timed automaton ps 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 in the recursion path. The `pt_remove` indicates that the states can be merged in the following recursion paths. Thus, this is the process to block them when merging each active event from s_2 in the corresponding events of s_1. An update for

Algorithm 4: `Deterministic_merge` function

input : A deterministic stochastic timed automaton $ps = (\mathcal{X}, \mathcal{E}, \Gamma, p, p_0, G)$ from sample executions starting in s, two states s_1 and s_2, a heap h, and a prefix tree t.

output: A deterministic stochastic timed automaton $\mathcal{M} = (\mathcal{X}, \mathcal{E}, \Gamma, p, p_0, G)$.

if $degree[s_1] > degree[s_2]$ **then** `state_swap`(ps, 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` ps`;`
`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 ps 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`;`
    foreach *event* e *in* $\tau(s, s_2)$ **do**
        if $e \in \tau(s, s_1)$ **then**
            `deterministic_merge`$(ps, s_1\, e, s_2\, e, h, t)$;
        else
            `psa_insert`(ps, 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**
        `deterministic_merge`$(ps, first[tuple], second[tuple], h, t)$;

state s_1 of states from ps that were pointing to state s_2 is needed. 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. The `pt_get_merged` function gets the equivalent state at time of the current recursion.

Remark 4.9. Note that we denote some auxiliary functions in algorithm 4 for known data structures. The `pt_get`, `pt_find`, `pt_add`, and `pt_remove` are auxiliary functions for the 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` are auxiliary functions for a classical heap.

4.2.3 Model selection applied to the generalized semi-Markov process

To conclude the learning method, we need to introduce the concept of distribution discriminant and its selection criteria. With a merged prefix tree, we acquire the parameters for distributions that better fits the sample data. This is done by the maximum likelihood estimator (MLE) and with a selection criteria [Soong, 2004, p. 277], which allows selecting the distribution with maximal log likelihood. To test the validity of the selection model, a fitting test could be applied [Stewart, 2009, p. 630], but also ensuring a fast convergence of the test will be a good principle. The convergence techniques are used to know if the sample size is enough or not to make a rigorous fitting. We adopt the method proposed by Dey and Kundu [2009] as a fundamental guide to solve the learning process with reliability analysis.

We propose the algorithm 5 to solve the estimation of distribution parameters using the MLE for Exponential, Weibull and Log-Normal distributions, and the log likelihood criterion as maximal value to select the better model. Other continuous probabilistic distributions, like Rayleigh, normal (with non negative values) and other continuous distributions can be considered further.

Considering a set of samples from a GSMP as denoted in the chapter 2, the model selection criterion based on the log-likelihood from each event and each distribution to test $\mathcal{L}_{Dist}$ is denoted by

$$ln\left[\mathcal{L}_{Dist}\left(\theta|x_1,...,x_n\right)\right] = \sum_{i=0}^{n} ln\left[f_{Dist}\left(x_i|\theta\right)\right] \tag{4.5}$$

where θ is a set of parameters, $\mathcal{L}_{Dist}$ is a distribution to calculate the log-likelihood, and $x_1, ..., x_n$ the sample clocks.[4]

[4]Please note that one event has associated only one distribution.

Algorithm 5: Estimation function

input : A deterministic stochastic timed automaton model *ps* from sample executions starting in s.
output: A deterministic stochastic timed automaton model.

```
for n ← 1 to |Q| do
    if removed[node[n]] = 0 then
        foreach event e in τ(s, node[n]) do
            parameters[0] = infer_Exponential(clocks[node[n] e]);
            parameters[1] = infer_Weibull(clocks[node[n] e]);
            parameters[2] = infer_Log-Normal(clocks[node[n] e]);
            switch max(log_Likelihood(parameters)) do  // Select distribution;
                case 1
                    dist[node[n]] ="Exponential";
                case 2
                    dist[node[n]] ="Weibull";
                case 3
                    dist[node[n]] ="Log-Normal";
```

The MLE is used to estimate the distribution parameters. After that we apply the log-likelihood to decide which is the best model to the sample data with the following formula.

$$ln\left[\mathcal{L}_{Dmax}\right] > max\{\forall_{Dist \neq Dmax}\, ln\left[\mathcal{L}_{Dist}\right]\} \tag{4.6}$$

The model acquired by the formula 4.6 is denoted by $Dmax$ distribution. Two or more distributions are used to calculate the likelihood value. They are the empirical distribution based on the samples, and the other distribution produced by the estimated parameters. Statistically, this means that one estimated distribution is more similar to the original distribution which has generated the samples to learning. Figure 4.2 depicts a comparison between estimation of distribution parameters. The blue line is obtained from 100 samples of $X \sim Weibull(\lambda = 1, k = 1.5)$. The green line has the estimated parameter $\lambda = 0.88$, and a log-likelihood of -87. The red line has the estimated parameters $\lambda = 0.98$ and $k = 1.5$, and a log-likelihood of -75.27. The yellow line has a log-likelihood of -75.73.

However, the chosen method does not have a sufficient decision criterion. It has to be complemented also with fit tests like the Kolmogorov-Smirnov or chi-square (X^2). After that step a good distributions can be obtained. The goodness of fit test is used to make a hypothesis test if distribution follows a certain distribution or not. The p-value and distances are given by this test.

For exponential distribution, the following situations occur:

$$\mathcal{L}\left(\lambda\right) = \lambda^n exp\left(-\lambda n \overline{x}\right) \tag{4.7}$$

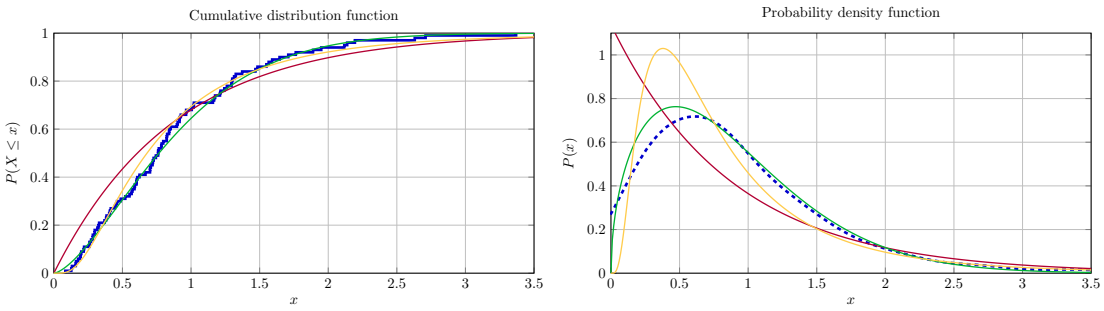


Figure 4.2: Graphical comparison between an empirical CDF (blue) and three estimated CDF (green, red and yellow).

$$ln\left[\mathcal{L}\left(\lambda\right)\right] = \sum_{i=0}^{n} ln\left[\lambda\, exp\left(-\lambda x_i\right)\right] \tag{4.8}$$

where equation 4.7 show the probabilistic density function of a Exponential distribution, and 4.8 is the log-likelihood function to calculate its likelihood.

It should be clear that the rate of GSMP in this process is equal to one. However, assuming that we need to learn the GSMP with unknown rate or different to one, in this case we need to use a more generalized process to estimate the parameters. So, we propose using the expectation-maximization algorithm which observes the rate plus clock value (i.e., labeled as *old clock*) as unobserved or an erroneous measure.

4.3 Correctness of our learning methodology

In order to show the correctness of our algorithm, we need to show that the GSMP that the learning algorithm produces is equivalent (in some sense) to the model that was used to generate the samples. Thus, our correctness process is subdivided in two parts. First, we need to check that a set of samples for the learning algorithm is a *structurally complete sample* (SCS). Second, ensuring a SCS, we need to prove that, in the limit, the error of merging two non equivalent states tends to zero. With these two parts we can prove that the model that is learned by the algorithm, in the limit, behaves as the original.

A SCS is a sample composed by a set of paths that explores every transition and every state. This solves a common problem known as *insufficient data training* to learn an equivalent model.[5] With a SCS we ensure that all information needed to learning a model is achieved.

[5]Only paths of infinite size guarantees that for any model, the learned model eventually converge to an equivalent.

The errors that PSS algorithm could make are now described. We have two types of errors: type I error (α) (where we reject the equivalence of states, when in fact it should be done) and type II error (β) (where we do not reject the equivalence of states, when in fact we should have done), to ensure that our algorithm decides correctly we need to reduce both errors to zero (see A.2 for more details).

Proposition 4.10. 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.

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

More details about the proposition that we present here can be seen in Yu [1971] and Klotz [1967]. Other related work to guarantee the bi-simulation of labeled Markov chains is given by Danos et al. [2006]. Our method does not consider the co-algebraic methods (it is based on a statistical approach). Defining the GSMP with these methods is complex due to GSMP being considered analytically intractable.

4.4 Abstractions of discrete event systems

The aerospace and automotive industry typically uses computer systems to simulate continuous systems. For instance, a satellite uses an on-board computer to calculate the corrections to maintain itself in the desired orbit. These type of systems are hard to run in real time due to their complexity. Thus, we propose here an abstraction for continuous systems (small-case complexity) to simulate these systems as discrete event systems. With this, we can verify and simulate these systems in a simple fashion (from the literature we know that there are several parallel algorithms that simulate DES, which increase the simulation performance).

There are open problems related to the representation of continuous systems in an event oriented model like DES, such as the conversion of continuous systems to discrete event systems and the attribution of events as time changes. Thus, while event oriented models can be expressed in terms of the DEVS modeling formalisms, there are continuous-variable system (CVDS) models that do not seem to have an equivalent representation in

event oriented models. This opens the question of how time-driven model can be converted to an event-driven model? It is possible for all models? We try to describe at least one solution for a small class of CVDS.

Nutaro [2005] has proposed a discretization algorithm to simulate ordinal differential equations (ODE) as DES (was refereed in 3.2). This method uses a discrete step value for segmenting continuous variables in discrete segments. For instance, if we have a variable between $[10, 22]$ (a continuous state space), the discretization with step of 1 produces the set of states $\mathcal{X} = \{10, 11, 12, ..., 22\}$ (a discrete state space). This type of uniform discretization is called as quantized state system solver (QSS1, QSS2, and QSS3) as introduced by Cellier et al. [2007]. There are several comparative studies in the literature between these algorithms, which show that the algorithms have notoriously different performance. These methods solve ODE and simulate them as discrete state changes.

Remark 4.11. The presented DES definition in chapter 2 is rather different from the definition of DEVS. DEVS aim at the discretization of continuous systems, but with a really set of restrictions. On the other hand, DES allow a much better abstraction for stochastic discrete event systems. Our goal is to explore the stochastic process that behaves as event change and does not as state change (respectively, DES and DEVS). Moreover, note that for SDES we establish an approach to verify it with a statistical model checker and for stochastic DEVS there is no such approach. For DEVS verification there are other approaches like Hwang and Zeigler [2009] and Cicirelli et al. [2010].

4.4.1 Comparing discrete event specification with stochastic timed automaton

The comparison between DEVS and DES is essential to understand their differences. From Zeigler et al. [2000] we know that DEVS can be seen as an extension of the Moore machine formalism, which is a finite state automaton where the outputs are only determined by the current state (and do not depend directly from the input and transitions). This extension is proposed in the two following phases:

- associating a lifespan with each state, and second
- providing a hierarchical concept with an operation, called coupling.

We present the formal definition of the atomic DEVS in order to explain shortly the implications of the approach proposed in this thesis. There is also DEVS with stochastic clocks called stochastic DEVS (STDEVS) [Castro et al., 2009], which will not be presented here in detail. Thus, DES are essential to simplify and check continuous dynamic systems, even in timed automaton or DEVS (or even stochastic timed automaton or STDEVS).

We will explain how DEVS work comparatively to the definitions of the timed automaton model exposed in chapter 2. In definition 10, we expose a model with three main parts: the input events (the inputs of the system), the internal behavior model (states change without any input due to their dynamics), and the output events (output change as state change). An occurrence of an external event changes the actual state to another state, instantaneously, and the internal dynamics keeps the actual clock values for the next state change.

We expose two definitions of DEVS, the first is the atomic DEVS and the second the stochastic DEVS. These definitions are defined according to Zeigler et al. [2000].

Definition 10. The atomic DEVS model is defined by $M = (\mathcal{X}^{\cdot}, \mathcal{Y}^{\cdot}, \mathcal{S}, \tau, \delta_x, \delta_\tau, \lambda)$ where

- $\mathcal{X}^{\cdot}$ is a set of input events,
- $\mathcal{Y}^{\cdot}$ is a set of output events,
- $\mathcal{S}$ is a set of states (that can be a infinite number of states),
- $\tau : \mathcal{S} \to \mathbb{R}_{[0,\infty]}$ is the time advance function where $\mathbb{R}_{[0,\infty]}$ is a set of non-negative real number with infinity,
- $\delta_x : Q \times \mathcal{X}^{\cdot} \to S$ is the external state transition function, where $Q = \{(s, e) | s \in \mathcal{S}, 0 \leq e \leq \tau(s)\}$ is the total states set and e is the elapsed time at s,
- $\delta_\tau : \mathcal{S} \to \mathcal{S}$ is the internal state transition function, and
- $\lambda : \mathcal{S} \to \mathcal{Y}^{\cdot}$ is the output function.

Definition 11. A STDEVS model has the structure $M_{ST} = (\mathcal{X}^{\cdot}, \mathcal{Y}^{\cdot}, \mathcal{S}, \tau, \mathcal{G}_{int}, \mathcal{G}_{ext}, P_{int}, P_{ext}, \lambda)$ where

- $\mathcal{X}^{\cdot}, \mathcal{Y}^{\cdot}, \mathcal{S}, \tau, \lambda$ have the same definition as in DEVS,
- $\mathcal{G}_{int} : S \to 2^{\mathcal{S}}$ is a function that assigns a collection of sets $\mathcal{G}_{int}(s) \subseteq 2^{\mathcal{S}}$ to every state s,
- $\mathcal{G}_{ext} : \mathcal{S} \times \mathbb{R}_{[0,\infty]} \times \mathcal{X}^{\cdot} \to 2^{\mathcal{S}}$ is a function that assigns a collection of sets $\mathcal{G}_{ext}(s, e, x) \subseteq 2^{\mathcal{S}}$ to each triplet (s, e, x),
- $P_{int} : S \times 2^S \to [0, 1]$ is a function that returns a probability $P_{int}(s, G)$ of the next internal state change, when the system is in state s, and carried in the internal transition set $G \in \mathcal{G}_{int}(s)$, and
- $P_{ext} : \mathcal{S}^{\cdot} \times \mathbb{R}_{[0,\infty]} \times \mathcal{X}^{\cdot} \times 2^S \to [0, 1]$ is a function that returns a probability for the next state given $\mathcal{G}_{ext}(s, e, x)$ that contains all the subsets of S, where s is a state, e is an elapsed time, and x is an event identifier when an external event arrives.

We invite the reader to read Zeigler et al. [2000] and Castro et al. [2009] for more details about DEVS and STDEVS, respectively. We describe next the differences between DEVS and timed automata and also the similarity between STDEVS and stochastic timed automata.

- The DEVS have a particular structure that allows us to model them as known Moore machines. It has a more intuitive basis properly than DES. So, we can support input occurrences that are external to the system (called input events, $\mathcal{X}_{\cdot}$), and it has an internal event associated to each state (called output events, $\mathcal{Y}_{\cdot}$). The discrete event systems only have a set of events, they do not separate the events as input and output, meaning by this that they do not support input events.

- The time structure of DEVS is static and is associated to each state, i.e., each state has fixed lifetime. In timed automata each event has associated one holding time.

- In the STDEVS a stochastic process changes the state. In the SDES there is a stochastic clock. The event with minor value is the winner.

Now, we try to emphasize the relation between STDEVS and SDES. We know that our definitions in chapter 2 are more comprehensive nevertheless the DEVS are closer to the classical dynamic systems due to the definitions of the system behavior as well as system structure.

Converting STDEVS in stochastic timed automata. Our learning approach allows the translation of this models through sample executions. However, there is certainly an equivalence between the two models. We are convinced that the GSMP can model this type of behavior because it is a more comprehensive model. On the other hand, the problem of input events can be solved by the synchronization of processes. For instance, suppose that we have two synchronized models, one of which generate inputs and cause changes in the other process externally. A simple example are queues, where the input of one is the output of the other.

4.4.2 Discrete and stochastic abstraction approaches

With the massification of simulations of continuous systems we need to explore simple ways to make a similar simulations but with less resources. Moreover, the verification procedures and test generation are more explored and simplified. For this, we short by proposing an introduction about DEVS and STDEVS that is clearly different from the DES and SDES definitions. Thus, here we propose a methodology to model a small class of continuous systems using DES definitions and adopting polynomials for event specification. And what we mean is that we can use a piecewise of polynomials (defined by their coefficients) to simulate the model as a discrete event system and produce the output as continuous systems by the calculation of the first order or second order polynomials.

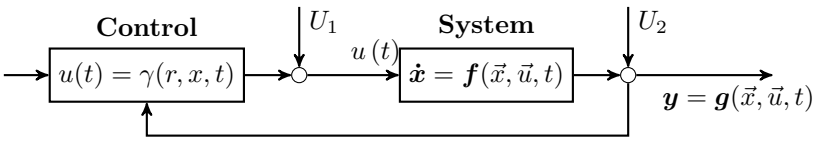


Figure 4.3: General diagram for the injection of disturbances in dynamical systems with feedback.

Perturbation models for dynamic systems. Perturbation models are widely used to test the dynamics of continuous systems. These models simulate several situations and certify that the continuous system can be used in this cases with success. The behavior of dynamic systems in some cases is tested with boundary tests that represent that the system is capable of modeling and controlling (i.e., the worst and optimal case execution of system). A typical diagram to inject perturbations is described in figure 4.3. This figure depicts the relation between a model (dynamic system) and a controller model (control system for dynamic systems).

Example 4.12. For instance, supposing that we have a cruise control system (controller) and an electrical accelerator (continuous system) we have the necessity to test this system in a set of cases. The potentiometer that measures the pedal position can give erroneous measures due to changes of humidity. Now, the system may not be tolerant for these measurement errors and crashes the continuous system. Indeed we are facing an uncertain cause that breaks the cruise control (a probable failure of system). We know that the failure of one potentiometer is controllable (humidity is linearly related to the resistance factor that the potentiometer measures). However, we do not know when a potentiometer can be broken due to an oxidization or other unknown cause.

In the following we describe two methods that solves these described problems. The first solution is to use test generation given a set of possible cases, using deterministic input models (e.g., for the linearly change of resistance due to humidity). The second one is using stochastic models to generate uncertainty in the input data set of a system (e.g., the resistor can be broken due to an oxidization or other unknown cause). For this two solutions are proposed DES and SDES, respectively. The first one allows to simulate and verify deterministic continuous dynamic models, and the second one allows a better and simple simulation and test. Moreover, we can use the stochastic models for verification, to ensure that a certain condition is allowed or not.

Deterministic perturbation models. We describe and exemplify a methodology for test and validation of dynamic systems that follows a common structure as presented in

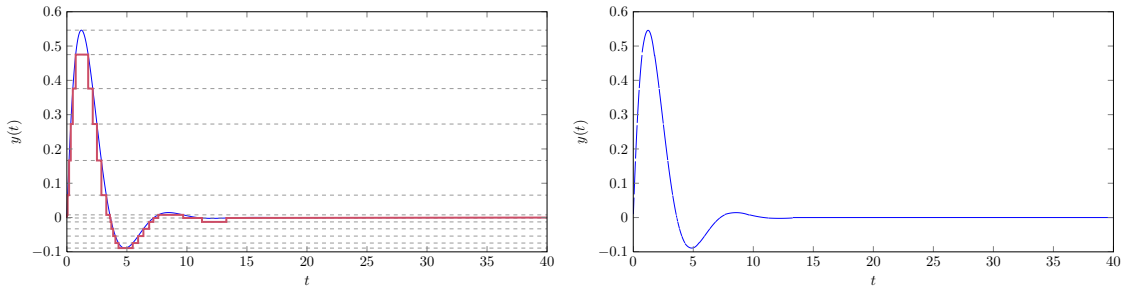


Figure 4.4: The graph (left) depicts the output of a second-order differential equation. The line (blue) is the simulation of this equation, and the line (red) is the quantization of this equation. Dashed lines are a non-uniform state space that are quantized by our algorithm. The graph (right) depicts a simulation of the discrete event system that abstracts the second-order differential equation.

the diagram of the figure 4.3. We will describe a small-scale example of the modeling of one perturbation model for an inverted pendulum.

Figure 4.3 depicts one dynamic system and its controller (like the cruise control example). We have two main blocks, the control block and the system block. The control block is defined by the function $\gamma(r, x, t)$, where r represents the scalar or a vector of reference for the controller (the desired equilibrium point for the controller), x the state vector of the system to control (given that is a system with closed-loop feedback), and t is a time variable. Also the dynamic system that is defined by $\dot{x} = f(\vec{x}, \vec{u}, t)$ is depicted, where x is the state vector of the system, and u the input vector for the dynamic system from the controller block.

The identifiers U_1 and U_2 are two perturbation models, one given by the measure noises and other by the control information noise. The difficulty is to find U_1 and U_2 such that they are appropriate to validate the system in a real way. Thus, we can directly apply the methods proposed in this thesis, for the acquisition of perturbation models (e.g., data given by a sensor network). Moreover, we can learn a model on very specific conditions (which depend on the dynamic system) and test the dynamic system in a realistic way. It should be noted that methods of pre-processing of data are necessary in order to obtain these models.

The figure 4.4 depicts one simulation of a particular second-order differential equation. We can see that with a DES that have 12 states and 25 events, we can produce a model that are really similar. The produced model by our learning algorithm is shows in figure 4.5. The estimated polynomials are depicted in the table 4.6.

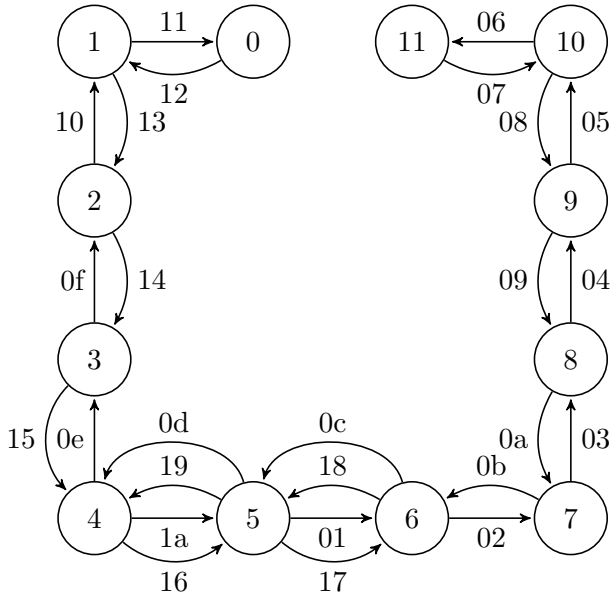


Figure 4.5: The discrete event system of the second-order differential equation illustrated in the figure 4.4 is depicted. It have 12 states and 25 events.

Second-degree polynomial coefficients			
Id	A	B	C
01	-0,49967	0,99998	1,1512e-07
02	-0,49615	0,99939	2,5477e-05
03	-0,48499	0,99504	0,00045
04	-0,46352	0,98063	0,00288
05	-0,42429	0,94036	0,01327
06	-0,26084	0,64487	0,14724
07	-0,08304	0,06473	0,62204
08	-0,01410	-0,23055	0,93849
09	0,03470	-0,47438	1,24328
0a	0,06673	-0,65782	1,50612
0b	0,07975	-0,74205	1,64234
0c	0,08150	-0,75433	1,66393
0d	0,08143	-0,75388	1,66309
0e	0,08059	-0,74755	1,65130
0f	0,07769	-0,72516	1,60794
11	0,07145	-0,67440	1,50475
12	0,04199	-0,40914	0,90762
13	0,00961	-0,06453	-0,00968
14	-0,00242	0,07800	-0,43166
15	-0,00991	0,17325	-0,73461
16	-0,01289	0,21375	-0,87222
17	-0,01311	0,21671	-0,88249
18	-0,00600	0,10279	-0,42612
19	0,00182	-0,04412	0,26420
1a	0,00099	-0,02403	0,14421

Figure 4.6: The table depicts the coefficients for each event. A event is composed by a 3-tuple of coefficients used to reconstruct the continuous behavior of the model.

Quantization and learning continuous systems. Dynamical systems typically evolve over a continuous state space, commonly called continuous-variable dynamic systems. We can verify this type of system with a low-level approach. So, methods for converting the state space are needed. The discretization should be made, with as little loss as possible so that the difference of the systems have an error e_{disc}, which is close to zero. Note that if there is a very small error this is probabilistically negligible, since the test of this hypothesis will be controlled.

The quantization starts by choosing the abstraction that is required. We begin by apply a clustering algorithm (K-means) in order to non-uniformly segment the continuous state space. After that we need to apply a polynomial fitting method in order to estimate the coefficients that are more close to the segment between thresholds of quantization. Applying that we can learn the path by labeling the discrete states. After this we have created a model like the one showed in figure 4.5.

Stochastic perturbation models. The stochastic injection models, as the name suggests, are models where the injection signals vary according to a stochastic process. With this perturbation model we can test or inject stochastically on continuous systems.

Given a set of continuous domain data we can quantize this and submit it to our learning algorithm. It produces a stochastic model based on events. However, the known

problem is too know how can throwback the process. We propose the use of polynomial chaos (stochastic polynomials) to better understand it.

Contextualizing the model checking, to verify a GSMP, we need to express this model in a specific language. This language is based on events. So, we have incorporated in our toolbox a converter according to the syntax defined in B. This language is directly accepted by Ymer [Younes, 2004] as input model, and it is equivalent to the input model language of PRISM [Kwiatkowska et al., 2011].

Optimizations in the implementation (as will be seen below) are made in order to model a large set of paths. This implementation was called by *SDES framework* and is developed for the Matlab environment. It allows the simulation of GSMP, its visualization, learning and testing. The tool was developed mainly in C and C++ languages.[6]

4.5 Model-based testing of stochastic discrete event systems

Model based testing is contextualized here in order to test deterministic systems and stochastic systems. Now, we propose two methods:

- We must have a model of DES (deterministic) that we can apply the inputs produced by a stochastic system and acquire the correct outputs from the classical model (the deterministic DES). So we have a set of unit tests based on a realistic source for any DES.

- Generate stochastic tests for a stochastic model

With appropriate perturbation models, we can validate the control system in a more realistic way, however the test of dynamic systems is not assured. The diagram of figure 4.7 illustrates the test for dynamic systems using GSMP, which aims to test the similarity of these systems. Suppose a SDES learned from an inverted pendulum system (through a discretization measurements taken), we can test its behavior according to the actual experiences, inferred through the learning process presented in this thesis. Thus, we can probabilistically decide whether the system acts in a certain range, correctly. The basic idea is equivalent to the one of Ševčíková et al. [2006] and Merayo et al. [2009], two approaches to test generation on stochastic and probabilistic systems. In Merayo et al. [2009] the authors proposed to create tests based on trajectories of stochastic and probabilistic finite state machines (SPFSM) and oracles of tests to ensure the satisfiability testing in stochastic and probabilistic finite state machines(SPFSM). Therefore, we propose as future work, the development of this test framework, which can either be based on tests derived

[6] http://desframework.sourceforge.net/

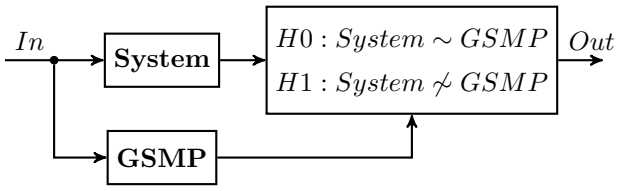


Figure 4.7: The diagram depicts the scheme for testing stochastic models with GSMP.

Language	files	blank lines	comment lines	code lines
C/C++ Header	24	403	501	3014
C	20	876	822	2338
MATLAB	16	395	544	2096
C++	9	422	289	962
Bourne Shell	1	115	141	789
Objective C	1	3	0	61
SUM	**71**	**2214**	**2297**	**9260**

Table 4.1: Code lines analysis of SDES toolbox for Matlab.

from GSMP, such as synchronized executions of both models (although the latter requires that the dynamic system and the GSMP be run simultaneously and synchronously).

4.6 SDES toolbox - Simulation, learning, verification and testing

One application that applies the previous learning definitions was developed. We created a small Matlab framework to analyze GSMP, learning GSMP, model conversion to event-driven languages, and a simple testing generation. The framework was created using the aid of two imperative languages C and C++. Both languages are interconnected with the Matlab engine and functions created with these languages are called in the Matlab console using the known Matlab language.

In table 4.1 we demonstrate an analysis of source code lines in order to give an overview of our Matlab framework interface. The framework has hundred thousand lines of code excluding the third-party code that our framework needs. The C language is the language that has the majority lines of code.

We have produced a set of practical examples, as described in the chapter 5, to show that our algorithm in practice is useful and scalable. We also developed a graphical user interface in order to simplify and make a user friendly interface. The figure 4.8 illustrates,

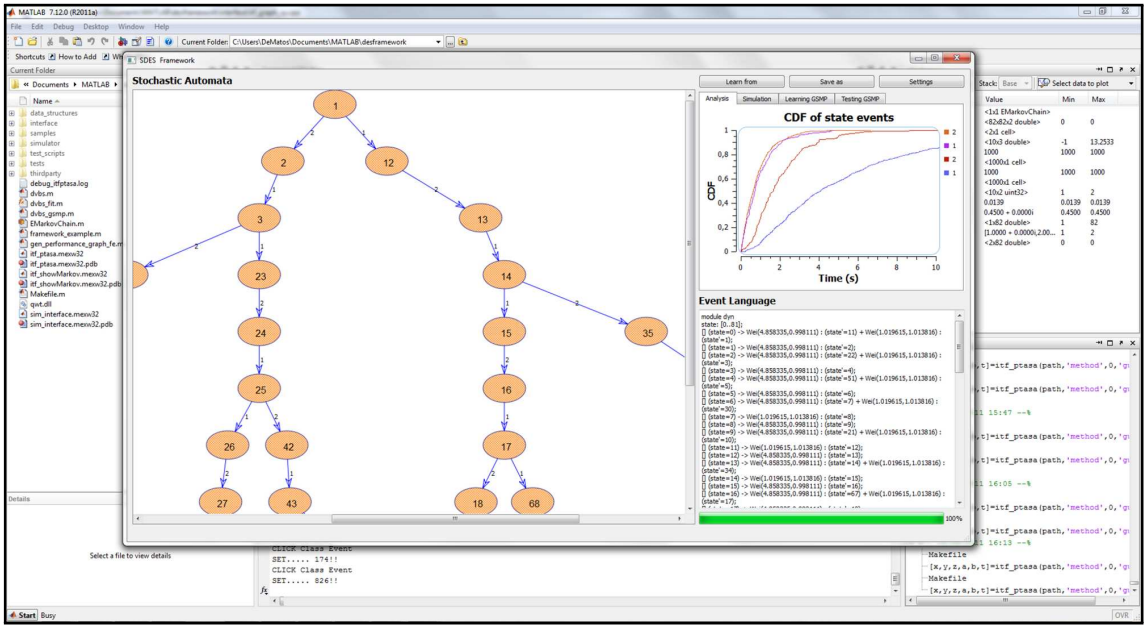


Figure 4.8: An overview of SDES framework with Matlab environment on background and in the top window our learning front end. The top window shows graphically the automaton (left), probabilistic distribution shape of selected event/s (top right), and the event-language production that corresponds to graphical automaton (top bottom).

without detail, the main window of the *SDES framework* inside Matlab environment. We can execute some useful tasks, like simulation, learning, testing and verification. The SDES framework could be used in a command line as function calls or with a simple graphical tool that shows graphically the distribution functions of each event. The interfaces for each function are described in appendix C. This GUI was developed due to a lack of Matlab to show graphically timed automata and stochastic timed automata. So, we created a Matlab diagram designer for Markov chains in a general way. The GUI is created with Qt toolkit for binding C++ language.

To understand how our framework works we made a high level diagram that approaches the previously theoretical presented solutions and the constructed application for Matlab. Diagram 4.9 shows the correspondence between applied methods and used data structures in it.

The learning process basically starts by generating a set of sample executions from a known GSMP model and simulating it. This simulation generates the sample executions that can be used for the learning algorithm. So, the essential of the learning algorithm are the Schedule Estimator (SE), Probabilistic Similarity of States (PSS), and lastly the Model Selection (MS). As we can see the SE receives a event-driven prefix tree constructed from sample executions, producing as output an event-driven prefix tree with clocks changed

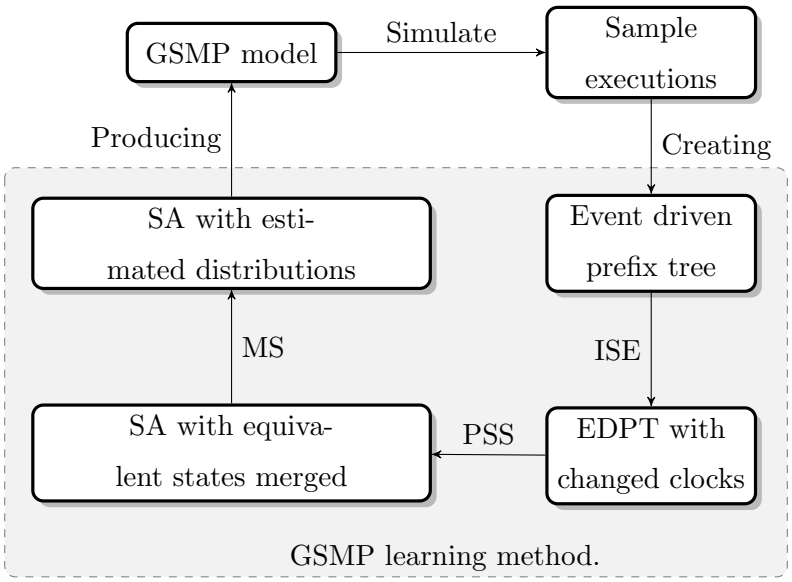


Figure 4.9: Diagram of learning a GSMP model from sample executions. The gray rounded rectangle show the three processes (SE, PSS, MS) involved in this model learning method and our interaction. The white rounded rectangles are the data structure involved in its particular step.

Figure 4.10: The diagram depicts the high-level sequence of the process to learn deterministic and stochastic continuous systems as SDES.

(to original samples). The PSS use its data structure to establish a stable relation between states. The event-driven prefix tree is a particular case of one stochastic automaton so we assume that and merge equivalent states. As output we produce a stochastic automaton. Now, we can apply the model selection in order to estimate the parameters of distributions for each event. Lastly, after these steps we can have one GSMP model similar to the original (the model recognizes at least the same language).

The perturbation models are quite different from the presented previously. The generation of perturbation models have the following process. First, we have a continuous state space and not a discrete state space that is a notorious difference. So, we need to setup an efficient mechanism to learn stochastic processes from a continuous dynamics. This mechanism is base on the application of stochastic polynomials between each discrete state. For that we need to apply a discretization. The diagram of the figure 4.10 illustrates how our toolbox makes it in practice.

In this chapter, we have proposed the learning approach for GSMP, the abstraction for continuous systems, and also two test generation approaches (i.e., use of stochastic models to generation of unit tests and stochastic test generation). In the next chapter we present practical case studies that take advantage of the methods proposed here.

Chapter 5

Evaluation of GSMP learning

In the previous chapter, we have proposed a learning approach, discrete abstractions for continuous systems, and two test generation approaches for generalized semi-Markov processes. In practice our proposal covers a set of practical case studies. Here, we evaluate three case studies. The first case study (see section 5.1) is an empirical analysis of a learned model from a set of sample executions generated from a known GSMP. We also compare the produced models with the variation of sample executions size. The second case study (see section 5.2) is an analysis with our learning algorithm for a DVB-S communication with a fast train, i.e., a land mobile satellite communication performance analysis. Lastly, we present a case study (see section 5.3) of stochastic linear abstractions for continuous systems.

We also show that our solution in second case study is more capable (automatic process) than other manuals methods (hand made process) used to produce the same result. The hand made process used by Sciascia et al. [2003], Scalise et al. [2008] to produce this case study are error prone which is clearly a disadvantage. Thus, using our learning methodology we can avoid these errors as we will explain in section 5.2.

The evaluations that we present in this chapter are meant as an aid to practitioners who want to use and applying GSMP learning. Our learning algorithm allows us to analyze the learned model and further if it is needed verify it by a statistical model checker like Ymer Younes [2004].

5.1 Learning from a known model: performance analysis

A practical analysis of the previously proposed method is presented in this section for two different empirical models. These two models were created to show in practice how our algorithm evolves, its performance, and also analyze how the outputs change according to

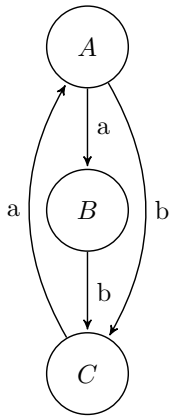


Figure 5.1: Empirical GSMP model as a stochastic automaton with three states and two events.

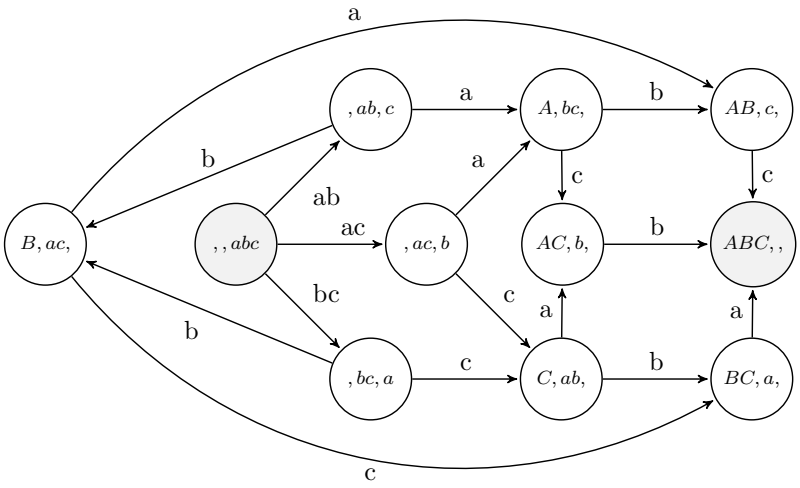


Figure 5.2: The empirical GSMP model of the two-core scheduler for three different tasks a, b, c. It is composed by 11 states and three CDF for each task, respectively, $Weibull(1, 1.5)$, $Exponential(0.8)$, and $Log\text{-}Normal(3, 0.9)$.

different sizes of the sample execution set. The first empirical model is a simple GSMP with three states and two events, which we use mainly to illustrate the scheduler behavior of the GSMP and its inverse estimation in our learning process. We describe also a performance analysis for the learned model with sample execution sets of different sizes and an analysis of the distribution parameters estimator for the learned clock structure. The second model was created to show an analysis of a two-processor system scheduler, in order to find a makespan optimal scheduler (i.e., the minor time difference between the start and finish of a sequence of tasks). We use this model to show that our method can produce the same output model and can be checked by a statistical model checker like Ymer [Younes, 2004]. Furthermore, we can use a similar approach in UPPAAL extended with statistical model checking [David et al., 2011]. At the moment of writing this thesis, the tool still has some limitations due to its recent development. The model checking allows the verification of one or more properties. For instance what is the probability that a scheduler (for two processors and three tasks) attributing the tasks in an order with less runtime?

Empirical model. The model $\mathcal{M} = (S, \mathcal{E}, f, \Gamma, x_0, G)$ is defined according the figure 5.1 by a set of three states $S = \{A, B, C\}$, two unique events $\mathcal{E} = \{a, b\}$, four transitions $f(A, a) = B,\ f(A, b) = C,\ f(B, b) = C,\ f(C, a) = A$, three active events set one for each

state $\Gamma(A) = \{a, b\}, \Gamma(B) = \{b\}, \Gamma(C) = \{a\}$, the initial state A, and lastly two PDF $G(a) = Exp(0.1)$ and $G(b) = Weibull(0.4, 1.25)$. This model stochastically identify the event language 'aba' or 'ba', as regular expression $([aba] + [ba])^*$.

Now, we describe one particularity of this GSMP given the definition of the stochastic clock structure (see section 2). Supposing that GSMP begins in state A and the event a is triggered we have $V_{a,1}$ and $V_{b,1}$ as initialized clocks with $Y^* = V_{a,1}$. The next triggered event is b so as $V_{b,1}$ was initialized we have $Y^* = V_{b,1} - V_{a,1}$.

Semi-empirical two-processors scheduler model. An optimal scheduler design for multi-processor systems with uncertain task duration is a difficult challenge [Ng et al., 2009, Pinedo, 2008]. We present a model in figure 5.2, from which it is possible to achieve statistically answers about worst case sequence and optimal case sequence of a two-processors scheduler system. The GSMP can run at the same time two concurrent tasks, so we need to know stochastically what is the more probable start sequence and further view what is the worst or optimal sequence. This GSMP is composed by eleven states and six events, three start events ab, ac, bc and other three events/tasks a, b, c. The empirical distributions to this model are $Exponential(1)$, $Exponential(1.15)$, $Exponential(1.25)$, for first three events and $Weibull(0.1, 1)$, $Exponential(0.4)$, and $Log\text{-}Normal(0, 0.25)$, respectively. So, the task to be processed first probabilistically is the ab, the second ac and lastly the bc.

We compare the performance of these two models, as the sample size grows. We present in the following, the steps to simulate and learn the two GSMP model from sample executions. For each GSMP we perform a discrete event simulation in order to get a large set of sample executions and then learning the GSMP with our algorithm. We ensure that the sample execution set is a structurally complete sample, i.e., all of the states are contemplated at least with one passage and all of the transitions are triggered at least one time in the set of sample executions. Lastly, we conclude that our algorithm allows the same stochastic event language as the original. The convergence of algorithm was analyzed with several simulations. We conclude that the samples are enough if the learning algorithm achieves, in different simulations, the same model.

We have made some tests in a machine with an *Intel Core 2 Duo CPU T7500 @ 2.2Ghz* with *4Gb* of memory. The results of our approach are shown in figure 5.3.

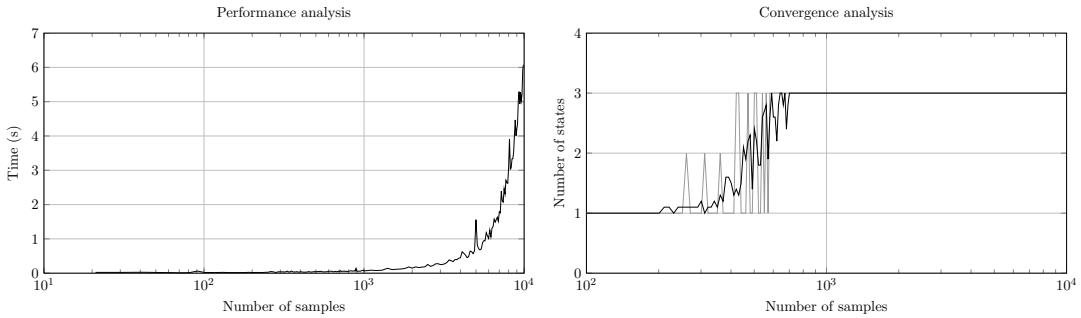


Figure 5.3: The performance and convergence evaluation of our method. The graph (left) is the performance analysis of our method to learn the previous empirical models, and the other graph (right) is the convergence analysis of our methodology.

5.2 Analysis of DVB-S communications for fast trains: a model

The railway environment constitutes an extremely challenging scenario for satellite based communications. As land mobile satellite communication (LMSC) [Csurgai-Horváth and Bitó, 2007], the railway satellite based communications (RSBC) is characterized also by three states: blockage state due to large obstacles like buildings, bridges , tunnels and train stations; shadowing state due to small or light objects like trees plus catenaries, electrical posts and trellises; and lastly LOS state when there is an absence of effects, the line of sight. For the long tunnels and train stations, proper gap fillers (e.g., terrestrial repeaters) have to be contemplated. The RSBC allows broadband Internet connections and multimedia services (e.g., digital TV) to be provided for passengers.

In this case study, we construct a model and we make a further analysis in order to show how useful is the proposed algorithm. So, we can verify some claims about our model as simple as saying that the occurrence of small obstacles is less than 0.15 (15%) or the occurrence of LOS is greater than 0.9 (90%). The data used in this case study is provided by a complete statistical analysis from Sciascia et al. [2003] and Scalise et al. [2008]. Through this analysis, we construct a sample generator in order to produce a similar dataset. With this dataset we show that the constructed model have the same particularities as the first order analysis provided by Sciascia et al. [2003], Scalise et al. [2008].

The main advantage in this case study using our methodology is the full automation of the modeling and verification process versus a hand made process. The hand made process is error prone. However, our learning algorithm ensures guaranties about the learning process, which it is the same as say that the process is not error prone.

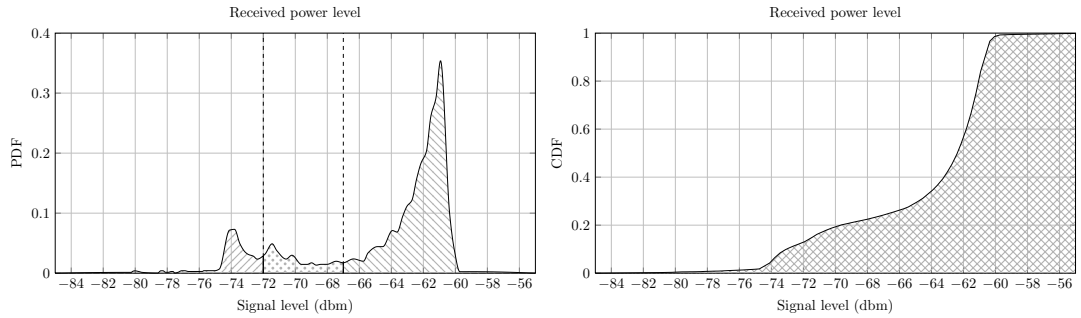


Figure 5.4: Graphs of PDF (left) and CDF (right) for the received power level of the land-satellite communication system in one trip from a high speed Italian railway (from Florence to Campiglia M.).

Reconstructing data samples. In this case study, we do not use a genuine dataset instead we use graphical plotted results from a statistical analysis from Sciascia et al. [2003] in order to produce a complete dataset. From a given distribution, we can approximately produce outputs like the original data samples. Given the PDF from figure 5.4, we can identify easily three peaks around -61dBm, -71.4 dBm and -74dBm where the distribution of the received signal level is more concentrated. This effect is due to some repetitive situations along the railway path. So, we can subdivide it in three segments like the above mentioned states: *shadowing*, *blockage* and *LOS*. The intervals $[-81dBm, -72dBm]$, $]-72dBm, -67dBm]$ and $]-67dBm, -56dBm]$ characterizes respectively the *blockage* state (north west lines), the *shadowing* state (dots) and *LOS* state (north east lines).

The samples are calculated using the *inverse transform sampling* (ITS) method. Let X be a random variable and F a cumulative distribution function. The ITS method starts by generating a random number u from the uniform distribution in the interval $[0, 1]$, and then computes the value x such that $F(x) = u$, where x is a sample from distribution F. The $F(x) = u$ can be used as inverse cumulative distribution function $X = F^{-1}(U)$, where X distribute F and $U \sim \mathcal{U}(0, 1)$.

After the reconstruction of states, we can estimate the holding time t_n for each event e_n in order to produce a sequence of events (e.g., *blockage* $\xrightarrow{e_1,t_1}$ *LOS* $\xrightarrow{e_2,t_2}$ *shadowing* $\xrightarrow{e_3,t_3}$ *LOS* $\xrightarrow{e_n,t_n} \cdots$). We know the fade time distribution and the non-fade time distribution, in figure 5.5, and also some time rates between *LOS* state and *blockage* state, and between *LOS* state and *shadowing* state. Applying this distributions to the estimated states we obtain the sample paths as described below. The probability of these three states are respectively 0.1023 (blockage state), 0.1399 (shadowing state), and 0.7578 (LOS state).

In general, the fading time and non-fading time are the time interval that is needed to cross in upper-ward (non-fading time) or down-ward (fading time) direction a threshold.

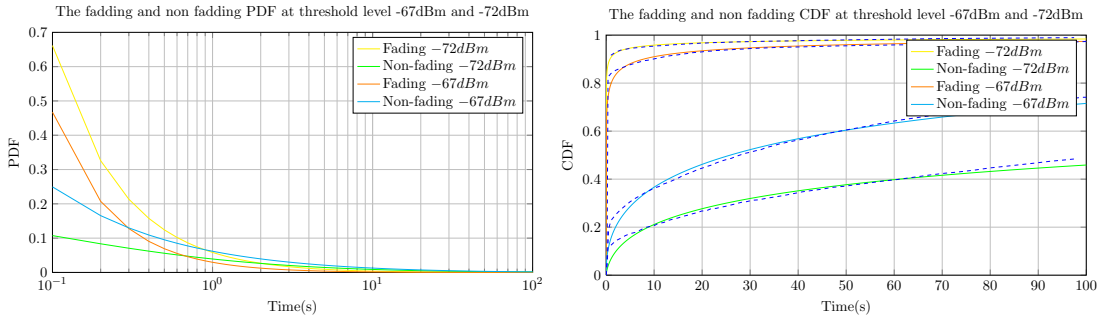


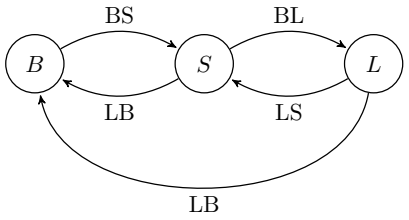
Figure 5.5: The PDF (left) and CDF (right) of the fading and non-fading distributions for two thresholds parameters respectively $-72dBm$ and $-67dBm$. The dash line (blue) represents the empirical PDF from original sample values, and the solid lines (yellow, blue, red and green) represent the estimated PDF (as we seen the estimated distribution are very close to empirical).

For example, supposing two thresholds $-72dBm$ and $-67dBm$, we know that non-fading time is the time interval that staying in the state *shadowing* crossing the $-67dBm$ and comes again in a downward direction to state *shadowing* or *blockage*. The fading time is precisely the inverse, and also the same happens to the $-72dBm$ threshold.

Learning the land-satellite communication model. The previously proposed learning algorithm for GSMP can be applied directly in this case study with small changes on data preprocessing. The learning algorithm basically consists of using set of sample executions to construct a SA without knowing the states. The samples collected from land-satellite communication system provide information about the label of states, but not of the identifier of events. As we seen previously, we know that in a DES there is a state space change when an event occurs. So, we consider an event when a state change occurs in the output measures of this system (e.g., change *blockage* state to *shadowing* state, *blockage* state to *LOS* state, etc.).

The sample executions from the realistic system (i.e., the land-satellite communication system) can be achieved directly by measures of signal level. Now, we have a sample path of 9600 seconds provided by the above reconstruction process. We know that the sequence of measures depends on time and it is needed to apply a simple preprocessing technique to label the events as the state of model changes. So, it finding the events and known the states, we need only to divide the path in a set of equal segments with a given size but always starting in the same state. After this step, the sample data can be learned by our learning algorithm and construct a land-satellite communication model of the Italian railway.

Analyzing the graphs of the figure 5.5 it is possible to view that there is an algebraic




```
1  module dvbs
2    state: [0..2];
3    [] (state=0) ->
4      L(-7.389563,2.872699) : (state'=1);
5    [] (state=1) ->
6      W(24.451307,0.459428) : (state'=0) +
7      L(3.729935,1.185887) : (state'=2);
8    [] (state=2) ->
9      W(24.451307,0.459428) : (state'=0) +
10     W(0.055173,0.187233) : (state'=1);
11 endmodule
```

Figure 5.6: The known model with three states and two events respectively W(5,1) and W(1,1).

composition between distributions with different thresholds. We describe the relation between distributions and events as follows: the non-fading distribution at $-72dBm$ represents the holding time distribution of event LB when state *LOS* (L) changes for state *blockage* (B); the non-fading distribution at $-64dBm$ represents the holding time distribution of event LS when state L changes for state *shadowing* (S); the fading distribution at $-72dBm$ represents the holding time distribution of event BS when state B changes for state S; and lastly the fading distribution at $-64dBm$ represents the holding time distribution of event BL when state B changes for state L. So, there are six events, the last four and two more events SB and SL as algebraic compositions from last ones. The SB is the difference between events LB and LS, and SL is the difference between events BS and BL.

For example, suppose the sequence of states B S B S L B S B S L the according events are respectively BS SB BS SL LB BS SB BS SL, where B is the start state and L the last state with respective initial event BS and final event SL.

Figure 5.6 illustrates the learned SA without event BL. This absence is due to the nonexistence of a passage from state B to L due to constrains from a signal level indicator of the common receivers. This is one interesting thing that our learning algorithm has detected. The box with code, shown in figure 5.6, an event language interpreted by the model checker Ymer (as referred in appendix B) that was represented graphically on left. We show in the following paragraph that the learned model has the same probability distributions than the ones that have generated them.[1]

Analyzing the model. In a long execution run we can verify that the probability that the model stays in LOS state is similar to the probability referred in the graph of the figure 5.4. The same occur for the other two states. We can prove properties like this one:

[1]Note that we show that our learning algorithm produces an equivalent model based on measured samples that produce the probabilistic distributions from the literature, obviously a real situation will be more appropriate.

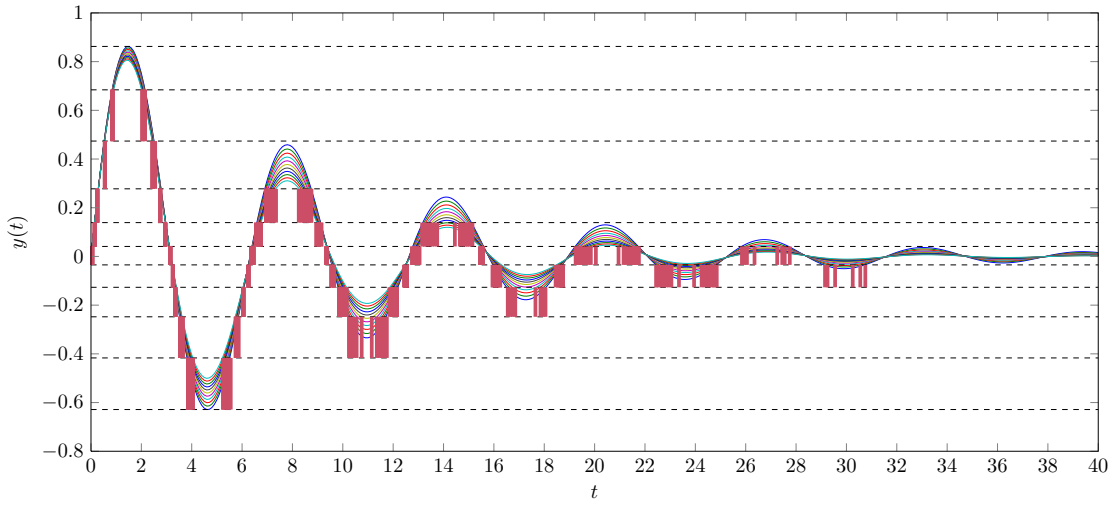


Figure 5.7: Discrete state space partition (the horizontal dotted lines) from 11 second-order differential equations, with states change interval (the red vertical lines).

$P_{\leq p}\,[true\,U^{96}\,LOS]$. It says that the model has one probability less than or equal to p of LOS is achieved until 96 seconds. However, this can be used to achieve the probability of event sequence of one system.

5.3 Learning a set of second-order differential equations as perturbation model to CVDS

We present here a simple example of a creation of a linear perturbation model for one particular CVDS (inverted pendulum). At the time of writing this thesis the stochastic polynomials were addressed as further work.

We explain now the figure 5.7. There are represented eleven second-order equations with different parameters. The goal is to produce a model that can abstract this behavior and produce linear random paths. We have learned the model with several paths. The dynamic system is simulated with hundred different parameters (randomly selected). Next, we have applied a non-uniform quantization in order to convert the continuous space state on a discrete space state. This can be viewed on several vertical (red) lines. Each state have different lifetimes. With this lifetimes we can estimate a probabilistic distribution with the help of our learning algorithm. Lastly, we have processed a set of paths to create the model of the figure 5.8. This model is a stochastic timed automaton. It is clearly a generalized semi-Markov process.

Range of states		
s_0	-0.6	-0.4
s_1	-0.4	-0.25
s_2	-0.25	-0.1
s_3	-0.1	-0.04
s_4	-0.04	0.02
s_5	0.02	0.14
s_6	0.14	0.3
s_7	0.3	0.48
s_8	0.48	0.66
s_9	0.66	0.86

Table 5.1: The boundary values of discretization of several second-order equation simulations.

Estimated parameters	
μ_1	3.01
μ_2	2.96
μ_3	3.43
σ_1	0.17
σ_1	0.19
σ_1	0.09
λ_1	0.73
λ_2	1.05
λ_3	4.85
k_1	6.89
k_2	5.68

Table 5.2: The estimated parameters of the learned (hundred) second-order simulations. It has eleven parameters.

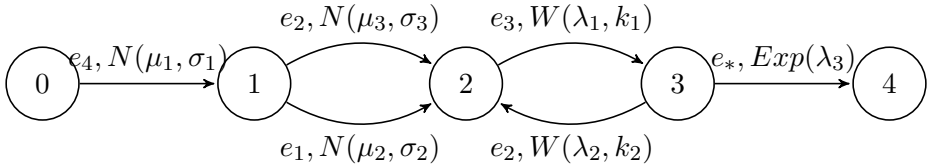


Figure 5.8: Stochastic automaton learned with our proposed method.

Non-Uniform discretization. This discretization is made with the help of a clustering algorithm (K-Means) like the method exemplified in section 4.4.2. In this case clustering the points between the range of -0.8 and 1 with a cluster parameter of twelve. We archive twelve states defined by the horizontal dashed lines (that are boundaries) of the figure 5.7. So, we have a state space with twelve states $\{s_1, s_2, s_3, s_4, ..., s_{12}\}$ that represents each pair of boundary values on table 5.1.

The values of table 5.1 are used and managed to reconstruct linearly the continuous domain. We describe here the two sample paths that exemplify the inputs for our algorithm that have generated the stochastic automaton of the figure 5.8. They are the following:

- $\phi_1 = s_5 \xrightarrow{e_1,t_1} s_6 \xrightarrow{e_2,t_2} s_7 \xrightarrow{e_3,t_3} s_8 \xrightarrow{e_4,t_4} s_9 \xrightarrow{e_5,t_5} s_{10} \xrightarrow{e_6,t_6} s_9 \xrightarrow{e_7,t_7} s_8...$
- $\phi_2 = s_5 \xrightarrow{e_1,t_1} s_6 \xrightarrow{e_2,t_2} s_7 \xrightarrow{e_3,t_3} s_6 \xrightarrow{e_4,t_4} s_5 \xrightarrow{e_5,t_5} s_4 \xrightarrow{e_6,t_6} s_3 \xrightarrow{e_7,t_7} s_4...$

We repeat this process recursively for several simulations (one hundred). Applying the *scheduler estimator* and the *probabilistic similarity of states* algorithms we obtain the model of the figure 5.8 without estimated parameters. The parameters of the table 5.2 are obtained using the model selection algorithm. With this information we can produce several random paths.

In this chapter, we have discussed three case studies. The first empirical evaluations

are judged to be successful and show that it can be applied to real scenarios like a real-time scheduler system tasks. Next, a realistic case study based on real data statistics revealed that our method is capable of the analysis of real-time systems. Lastly, we have described a stochastic approach to generate linear perturbation models. We have addressed the learning of polynomial stochastic perturbation models for further work, which were our goal initially. In the next chapter we conclude our thesis and propose further work.

Chapter 6

Conclusion and Future Work

At the outset of this thesis we embarked on an ambitious endeavor to develop algorithms for learning GSMP and abstractions for continuous systems. We believe our research effort to be a good start towards practical solution techniques for stochastic discrete event systems, but we most certainly acknowledge that we have only scraped the surface of this vast area of research.

In the area of learning stochastic discrete event systems, we have established an inclusion between this model and a more abstract model (generalized semi-Markov processes). We have implemented a new and the first algorithm to learn GSMP. This learning algorithm allows us to construct models given data from realistic sensor networks or even samples from real environments. With this we can provide a set of features such as the possibility of statistically verifying these models using statistical model checking and testing deterministic models or even creating stochastically a suite of tests. We have ensured that our learning algorithm, in the limit, is equal or similar to the one that was used for learning. We demonstrate the proposition that merging two equivalent states is correct when sample executions grow infinitely. We also show that the convergence of the Kolmogorov-Smirnov test is reachable (i.e., a exponential convergence). We also have proposed an algorithm to estimate the scheduler of events of a GSMP. This allows us to estimate the original clock values and estimate the parameters of the probabilistic distributions coupled to each event. A potential benefit of discrete event systems is that they tend to be highly amenable to parallelization in comparison to other common systems. We have exemplified one real case study that can be amenable for analysis and simulation. We can use this model to simulate the high speed train availability to the satellite and test a new land-satellite communication protocols. We have exemplified an analysis of scheduling algorithms for real-time systems when the uncertainty govern the execution time of tasks.

Our contribution to the abstraction of continuous systems addresses the conversion of dynamic systems (modeled by differential or difference equations) to the discrete event systems. We base this method on an established non-uniform quantization. Non-uniform quantization methods have been absent in quantization and simulation of ordinal differential equations. We have presented two approaches. The first for deterministic system like second-order differential equations and other not quite a complete solution but a beginning of one for stochastic systems. Thus, the discrete event system model for second-order differential equation has demonstrated that abstractions of continuous systems with non-uniform quantization and piecewise of polynomials are very accurate. On the other hand, this discrete event model could be simulated with less resources. Also we have described that, due to polynomial piecewise events, we can reduce drastically the size of state space of the system.

The complete stochastic test generation, as referred as a generic goal of thesis, was addressed partially. The completion of this task is clearly future work. We propose testing stochastically using the aid of deterministic models to generate a set of tests for common deterministic systems.

It is clear that there are systems in the real world for which our assumptions are inappropriate. Widening the extensions of our framework is definitely future work. We have provided practical techniques for learning generalized semi-Markov processes, abstractions of continuous systems and testing using stochastic models. We also have presented a set of evaluation case studies that are all solved with success. The possibility to verify statistically the learned stochastic models is a great advantage. Using an approximated model, amenable to numerical verification techniques, is generally hard to quantify the effect that a model approximation has on the validity of the verification result. With our approach this does not occur. As more sample are available more probable is that our model is similar to the unknown or original model (in the limit, the decisions that are made by our algorithm are correct, in the sense that they are indistinguishable from the original one).

In future research, we plan to identify several complex and real-world applications for the techniques that we have developed. We also address as future work the creation of a framework for test generation. We conclude by saying that our contributions have greatly expanded the actual state of the art of stochastic discrete event systems, thus allowing a much wider covering of systems such as the analysis of real models, verification of learned models, and testing them as realistic perturbation models.

Appendix A

Statistical background

A.1 Random number generators

An excellent reference for background on event simulation, random generation and statistical validation techniques is the book by Ross [2006]. The book describes Uniform distribution, Exponential distribution, Weibull distribution, and Log-Normal distribution generation. Another great reference is Soong [2004].

A.2 Statistical validation techniques - model verification

A statistical hypothesis test is a method of making decisions using data, whether from distributions that may be specified completely with prespecified values for their parameters or that may be specified with parameters yet to be estimated from the sample. We use the definition of Type-I and type-II errors according to Soong [2004]. We also expose one decision method, namely, the Kolmogorov-Smirnov test.

A.2.1 Preliminaries

Let $X_1, X_2, ..., X_n$ be an independent sample of size n from a population X with a hypothesized probability density function (pdf) $f(x;)$ or probability mass function (pmf) $p(x;)$, where may be specified or unspecified. We denote by hypothesis H the hypothesis that the sample represents n values of a random variable with pdf $f(x;)$ or $p(x;)$. This hypothesis is called a simple hypothesis when the underlying distribution is completely specified; that is, the parameter values are specified together with the functional form of the pdf or the pmf; otherwise, it is a composite hypothesis. To construct a criterion for hypotheses testing, it is necessary that an alternative hypothesis be established against which hypothesis H can be tested. An example of an alternative hypothesis is simply

another hypothesized distribution, or, as another example, hypothesis H can be tested against the alternative hypothesis that hypothesis H is not true.

In our applications, the latter choice is considered more practical and we shall in general deal with the task of either accepting or rejecting hypothesis H on the basis of a sample from the population.

A.2.2 Type-I and type-II errors

As in parameter estimation, errors or risks are inherent in deciding whether a hypothesis H should be accepted or rejected on the basis of sample information. Tests for hypotheses testing are therefore generally compared in terms of the probabilities of errors that might be committed. There are basically two types of errors that are likely to be made – namely, reject H when in fact H is true or, alternatively, accept H when in fact H is false. We formalize the above with definition 12.

Definition 12. In testing hypothesis H, a Type-I error is committed when H is rejected when in fact H is true; a Type-II error is committed when H is accepted when in fact H is false.

In hypotheses testing, an important consideration in constructing statistical tests is thus to control, insofar as possible, the probabilities of making these errors. Let us note that, for a given test, an evaluation of Type-I errors can be made when hypothesis H is given, that is, when a hypothesized distribution is specified. In contrast, the specification of an alternative hypothesis dictates Type-II error probabilities. In our problem, the alternative hypothesis is simply that hypothesis H is not true. The fact that the class of alternatives is so large makes it difficult to use Type-II errors as a criterion. In what follows, methods of hypotheses testing are discussed based on Type-I errors only.

The table of probability errors in a test hypotheses is described in the following.

Table A.1: Relations between truth/falseness of the null hypothesis and outcomes of the test.

	Null hypothesis (H_0) is true	Null hypothesis (H_0) is false
Reject null hypothesis	Type I error	Correct outcome
Fail to reject null hypothesis	Correct outcome	Type II error

A.2.3 Kolmogorov-Smirnov

The Kolmogorov-Smirnov (K-S) test [DeGroot, 1989, Stewart, 2009, p. 552,p. 625] is used to detect differences between two distributions. We use this test for the algorithm 2.

Let $\{X_n\}_{n\geq 1}$ and $\{Y_n\}_{n\geq 1}$ be two independent successions of independent real random variables with common distribution functions, respectively F_1 and F_2. The Kolmogorov-Smirnov test allows testing,

$$\begin{aligned} H_0 : F_1(x) &= F_2(x), \text{for all } x \in \mathbb{R} \text{ against} \\ H_1 : F_1(x) &\neq F_2(x), \text{for some } x \in \mathbb{R} \end{aligned} \tag{A.1}$$

using the statistic test

$$T_{n_1,n_2} = \sqrt{\frac{n_1 n_2}{n_1 + n_2}} \underset{x\in\mathbb{R}}{sup} |F_{n_1}(x) - F_{n_2}(x)| \tag{A.2}$$

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 real random variable T_{n_1,n_2} converges into lay of Kolmogorov-Smirnov with distribution function,

$$G(t) = \left(1 - 2\sum_{i=1}^{\infty}(-1)^{i-1}\ exp(-2i^2t^2)\right)\ \mathbb{I}_{]0,+\infty[}(t) \tag{A.3}$$

whose values are tabled in [DeGroot, 1989, 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 $G(k_\alpha) = 1 - \alpha$.

Appendix B

Event language

This section presents the syntax of a module like Ymer input language, characterizing a GSMP. The extended BNF of Ymer input language have the following conventions:

- Each rule is of the form ⟨*non-terminal*⟩ ::= expansion.
- Alternative expansions are separated by a vertical bar ("|").
- An asterisk ("*") following a syntactic element x means zero or more occurrences of x.
- Terminals are written using `typewriter` font.
- Case is significant. For example, X and x are separate identifiers.
- Parentheses and square brackets are an essential part of the syntax and have no semantic meaning in the extended BNF notation.
- Any number of whitespace characters (space, newline, tab, etc.) may occur between tokens.

There are two top-level syntactic elements that may occur in an input file: ⟨*model*⟩ and ⟨*property*⟩. A ⟨*name*⟩ is a string of characters starting with an alphabetic character followed by a possibly empty sequence of alphanumeric characters, hyphens ("-"), and underscore characters (" "). A ⟨*pname*⟩ is a name immediately followed by a prime symbol ("'"). An ⟨*integer*⟩ is a non-empty sequence of digits. A ⟨*number*⟩ is a sequence of numeric characters, possibly with a single decimal point (".") at any position in the sequence, or two integers separated by a slash "/". A ⟨*probability*⟩ is a number with a value in the interval [0, 1].

B.0.4 BNF

⟨*model*⟩ ::= ⟨*model-type*⟩ ⟨*declaration*⟩* ⟨*module*⟩*

⟨*model-type*⟩ ::= stochastic | ctmc | gsmp

⟨*declaration*⟩ ::= const ⟨*name*⟩ = ⟨*integer*⟩;

| rate ⟨*name*⟩ = ⟨*number*⟩ ;

| global ⟨*name*⟩ : ⟨*range*⟩ ;

| global ⟨*name*⟩ : ⟨*range*⟩ init ⟨*expr*⟩ ;

⟨*range*⟩ ::=[⟨*expr*⟩ .. ⟨*expr*⟩]

⟨*module*⟩ ::= module ⟨*name*⟩ ⟨*variable-decl*⟩* ⟨*command*⟩* endmodule

| module ⟨*name*⟩ = ⟨*name*⟩ [⟨*substitution-list*⟩] endmodule

⟨*substitution-list*⟩ ::= ⟨*name*⟩ = ⟨*name*⟩ | ⟨*name*⟩ = ⟨*name*⟩ ,⟨*substitution-list*⟩

⟨*variable-decl*⟩ ::= ⟨*name*⟩ : ⟨*range*⟩ ;

| ⟨*name*⟩ : ⟨*range*⟩ init ⟨*expr*⟩ ;

⟨*command*⟩ ::= ⟨*synchronization*⟩ ⟨*formula*⟩ -> ⟨*distribution*⟩ : ⟨*update*⟩ ;

⟨*synchronization*⟩ ::= [ ] | [⟨*name*⟩]

⟨*formula*⟩ ::= ⟨*formula*⟩ & ⟨*formula*⟩ | ⟨*formula*⟩ | ⟨*formula*⟩ | ! ⟨*formula*⟩

| ⟨*expr*⟩ ⟨*binary-comp*⟩ ⟨*expr*⟩ | (⟨*formula*⟩)

⟨*binary-comp*⟩ ::= < | <= | >= | > | = | ! =

⟨*distribution*⟩ ::= ⟨*rate-expr*⟩

| Exp (⟨*rate-expr*⟩)

| W (⟨*rate-expr*⟩ , ⟨*rate-expr*⟩)

| L (⟨*rate-expr*⟩ , ⟨*rate-expr*⟩)

| U (⟨*rate-expr*⟩ , ⟨*rate-expr*⟩)

⟨*update*⟩ ::= ⟨*name*⟩ = ⟨*expr*⟩ | ⟨*update*⟩ & ⟨*update*⟩ | (⟨*update*⟩)

⟨*expr*⟩ ::= ⟨*integer*⟩ | ⟨*name*⟩ | ⟨*expr*⟩ ⟨*binary-op*⟩ ⟨*expr*⟩ | (⟨*expr*⟩)

⟨*binary-op*⟩ ::= + | - | *

⟨*rate-expr*⟩ ::= ⟨*integer*⟩ | ⟨*name*⟩ | ⟨*rate-expr*⟩ ⟨*rate-op*⟩ ⟨*rate-expr*⟩ | (⟨*rate-expr*⟩)

⟨*rate-op*⟩ ::= * | /

Appendix C

A Matlab interface of SDES toolbox

Our application has a set of Matlab functions. Now, we describe the interface of these functions followed by a brief description of the input/output data types. Our toolbox has more functions that are not described here. They are experimental functions for perturbation model generation, testing GSMP, and some examples for Matlab. Those examples are generated automatically for the case studies presented in chapter 5 (at the present moment there are three case studies).

C.1 Examples

To understand how the GSMP could be learned with our toolbox we made some exemplifications that we will present below. So, adopting the Matlab code of the figure C.1 we show the creation of a stochastic automaton that is composed by three states declared in `Mc.S` with respective labels in `Mc.Sl`, two events declared in `Mc.E` with also respective labels `Mc.El`, the boolean matrix `Mc.ES` that indicates the active events in each state, the 3-dimensional p matrix with probability of transition from state s to state s' given the event e, and finally the definition of parameters for each distribution of events. We also show in figure C.1 its graphic representation to understand easily the definition of the GSMP. We can easily use this model for simulation, producing sample executions, which can be applied on a further learning process. For example as follows:

```
>> [hsclk symbpath] = SDES_simulator(MC.S,MC.E,MC.ES,MC.p,MC.G, 10, seed);
```

or simply as,

```
>> [path] = SDES_simulator(MC, 10, 'path', 1);
```

to simulate the showed example. Given a set of paths from several simulations (executing

```
MC = MarkovChain_handler;

MC.S = [1,2,3];
MC.Sl = {'s1', 's2', 's3'};
MC.E = [1,2];
MC.El = {'t1','t2'};
MC.ES = [1 1; 1 0; 0 1];
MC.p(:,:,1) = [0 0 1; 1 0 0; 0 0 0];
MC.p(:,:,2) = [0 1 0; 0 0 0; 0 1 0];
MC.G = {'exp', {1}; 'wbl', {1,1}};

% Inverse the matrices
MC.p(:,:,1) = MC.p(:,:,1)';
MC.p(:,:,2) = MC.p(:,:,2)';

SDES_show(mc);
```

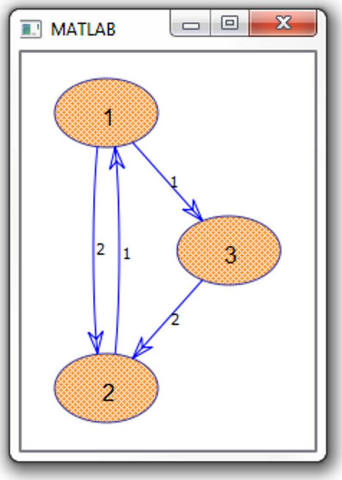


Figure C.1: Code example (left) and its respective Markov chain (right) that showing a simple example to understand the Markov chain declaration and visualization within Matlab.

`SDES_simulator` function) we can use these sample executions as input for the learning process.[1] The learning of the model can be made typing the following command:

```
>> [x,y,z,a,b,t] = SDES_psa(path);
```

or with,

```
>> [x,y,z,a,b,t] = SDES_psa(path,'method',0,'gui',1);
```

to aid graphically the learning process without applying any method, or also with,

```
>> [mc] = SDES_psa(path,'gui',1);
```

to aid graphically the learning process with the PSS algorithm applied. The outputs `[x,y,z,a,b,t]` are the learned model from a set of matrices as, respectively, the states matrix, the events matrix, the active events 2-dimensional matrix, the probability 3-dimensional matrix, the matrix that includes the distribution parameters of clocks, and lastly the time spent in the learning process. The graphic user interface (gui) is enough to debug or verify the overall process.

C.2 Alphabetical function list

C.2.1 Function 'SDES_psa'.

Purpose. This function learns a GSMP model from sample executions. For example, it can be used with a path structure as SDES_psa(*path*), where *path* is a cell array with dimension n that contains in each element two-dimensional matrix respectively with the event id and its time duration.

[1] Note that all paths can be inserted in a cell array before submitting it to the learning algorithm.

Description. The front-end of learning GSMP algorithm for Matlab. Using Qt to display windows and diagrams, and Graphviz to arrange graphically the nodes, the transitions and the labels before showing in Qt environment. As output produces a learned model from sample executions in two formats, first on a language interpreted by model-checker Ymer, and second, stores a set of matrices in the Matlab environment.

Syntax.

```
>> SDES_psa(path)
>> [stts,evts,E,P,G] = SDES_psa(path,'method',mid,'gui',guion)
```

Outputs

`stts`	The finite state set.
`evts`	The finite event set.
`E`	The binary matrix NxM of active events (N) for M states.
`P`	The 3-dimensional probability matrix (NxMxE).
`G`	The cell array of stochastic distributions.

Inputs

`path`	The path structure.
`mid`	Using '0' does not apply any method and '1' applies the PSS algorithm.
`guion`	A boolean variable to activate or deactivate GUI.

C.2.2 Function 'SDES_show'.

Purpose. This function graphically displays a Markov chain or any stochastic automaton from a set of matrices. For example, it can be used with a defined Markov chain object as SDES_show(mc).

Description. This function is implemented using the Qt toolkit to display any Markov chain or stochastic automaton in a window of Matlab. Note that Matlab has a lack of support for displaying graphs or classical automatons. So, we have implemented this function to view graphically the data from Markov chain object.

Syntax.

```
>> SDES_show(mc)
```

Outputs

Without functional outputs.

Inputs

`mc`	A Markov chain object handle.

C.2.3 Function 'SDES_simulate'.

Purpose. This function simulates a GSMP from a defined model and produces a path as output. For example, it can be used with one Markov chain as the first parameter and the number of steps to simulate the GSMP as the second argument; SDES_simulate_gsmp(mc, 10).

Description. This function implements the interface with the simulator for the GSMP. In other words this is a stochastic discrete event system simulator based on the relaxed Markov property. So, the simulator uses one scheduler to store the age of events.

Syntax.

```
>> [path] = SDES_simulate(mc,size)
```

Outputs

`path` The produced path from GSMP simulation.

Inputs

`mc` A Markov chain object handle.

`size` The number of steps to simulate.

Bibliography

Husain Aljazzar, Matthias Kuntz, Florian Leitner-Fischer, and Stefan Leue. Directed and heuristic counterexample generation for probabilistic model checking: a comparative evaluation. In *Proceedings of the 2010 ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems*, QUOVADIS '10, pages 25–32, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-972-5.

Husain Aljazzar, Florian Leitner-Fischer, Stefan Leue, and Dimitar Simeonov. Dipro: a tool for probabilistic counterexample generation. In *Proceedings of the 18th international SPIN conference on Model checking software*, pages 183–187, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22305-1.

S. Asmussen and P. W. Glynn. *Stochastic simulation: algorithms and analysis*. Springer, New York, 2007.

Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. ISBN 026202649X, 9780262026499.

Jerry Banks. *Handbook of Simulation : Principles, Methodology, Advances, Applications, and Practice*. Wiley-Interscience, September 1998. ISBN 0471134031.

Vlad Barbu and Nikolaos Limnios. *Semi-Markov Chains and Hidden semi-Markov Models toward Applications: Their Use in Reliability and DNA Analysis*. Springer Publishing Company, Incorporated, 1 edition, 2008. ISBN 0387731717, 9780387731711.

Ananda Basu, Saddek Bensalem, Marius Bozga, Benoît Caillaud, Benoît Delahaye, and Axel Legay. Statistical abstraction and model-checking of large heterogeneous systems. Research Report RR-7238, INRIA, March 2010.

Jean Claude Carmona and Norbert Giambiasi. New design and simulation of the gdevs abstraction of an integrator. In *SCSC*, pages 331–338, 2007.

Rafael C. Carrasco and José Oncina. Learning stochastic regular grammars by means of a state merging method. In *Proceedings of the Second International Colloquium on*

Grammatical Inference and Applications, pages 139–152, London, UK, 1994. Springer-Verlag. ISBN 3-540-58473-0.

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.

Rafael C. Carrasco, Mikel L. Forcada, and Laureano Santamaría. Inferring stochastic regular grammars with recurrent neural networks. In *Proceedings of the 3rd International Colloquium on Grammatical Inference: Learning Syntax from Sentences*, pages 274–281, London, UK, 1996. Springer-Verlag. ISBN 3-540-61778-7.

Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 0387333320.

Rodrigo Castro, Ernesto Kofman, and Gabriel A. Wainer. A formal framework for stochastic devs modeling and simulation. In *Proceedings of high performance computing symposium (HPCS)*, pages 421–428. (SCS) : The Society for Modeling and Simulation International, ACM Press, (SCS) : The Society for Modeling and Simulation International, February 2009.

François E. Cellier, Ernesto Kofman, Gustavo Migoni, and Mario Bortolotto. Quantized state system simulation, 2007.

F. Cicirelli, A. Furfaro, L. Nigro, and F. Pupo. Temporal verification of rt-devs models with implementation aspects. In *Proceedings of the 2010 Spring Simulation Multiconference*, SpringSim '10, pages 130:1–130:8, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0069-8.

László Csurgai-Horváth and János Bitó. Attenuation time series synthesis for land mobile satellite links. *In Proceedings 16th IST Mobile & Wireless Communications Summit*, 2007.

Vincent Danos, Josee Desharnais, François Laviolette, and Prakash Panangaden. Bisimulation and cocongruence for probabilistic systems. *Inf. Comput.*, 204(4):503–523, 2006.

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.

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. *INForum - SOFTPT*, 2011.

Morris H. DeGroot. *Probability and Statistics, 2nd edition.* Carnegie-Mellon University, Addison Wesley, 1989.

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.

Gordon Fraser, Franz Wotawa, and Paul E. Ammann. Testing with model checkers: a survey. *Softw. Test. Verif. Reliab.*, 19:215–261, September 2009. ISSN 0960-0833. doi: 10.1002/stvr.v19:3.

P. W. Glynn. A gsmp formalism for discrete event systems. *Proceedings of The IEEE*, 77: 14–23, 1989. doi: 10.1109/5.21067.

E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5): 447–474, 1967.

Tingting Han, Joost-Pieter Katoen, and Berteun Damman. Counterexample generation in probabilistic model checking. *IEEE Trans. Software Eng.*, 35(2):241–257, 2009.

Mor Harchol-Balter and Allen B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Trans. Comput. Syst.*, 15:253–285, August 1997. ISSN 0734-2071.

Robert M. Hierons and Mercedes G. Merayo. Mutation testing from probabilistic and stochastic finite state machines. *J. Syst. Softw.*, 82:1804–1818, November 2009. ISSN 0164-1212. doi: 10.1016/j.jss.2009.06.030.

Moon Ho Hwang and Bernard P. Zeigler. Reachability graph of finite and deterministic devs networks. *IEEE T. Automation Science and Engineering*, 6(3):468–478, 2009.

Sumit Kumar Jha, Edmund M. Clarke, Christopher James Langmead, Axel Legay, André Platzer, and Paolo Zuliani. A bayesian approach to model checking biological systems. In *CMSB*, pages 218–234, 2009.

David G. Kendall. Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain. *The Annals of Mathematical Statistics*, 24(3):338–354, 1953. ISSN 00034851. doi: 10.2307/2236285.

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. ISBN 3-540-44239-1.

Jerome Klotz. Asymptotic efficiency of the two sample Kolmogorov-Smirnov test. *Journal of the American Statistical Association*, 62(319):932–938, 1967.

M. Kwiatkowska, G. Norman, D. Parker, and J. Sproston. *Modeling and Verification of Real-Time Systems: Formalisms and Software Tools*, chapter Verification of Real-Time Probabilistic Systems, pages 249–288. John Wiley & Sons, 2008.

Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In *CAV*, pages 585–591, 2011.

Axel Legay, Benoît Delahaye, and Saddek Bensalem. Statistical model checking: An overview. In *RV*, pages 122–135, 2010.

Will Leland and Teunis J. Ott. Load-balancing heuristics and process behavior. *SIGMETRICS Perform. Eval. Rev.*, 14:54–69, May 1986. ISSN 0163-5999.

Daniel Lowd and Jesse Davis. Learning Markov network structure with decision trees. In *Proceedings of the 2010 IEEE International Conference on Data Mining*, ICDM '10, pages 334–343, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4256-0.

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*, Springer Series in Reliability Engineering, pages 201–210. Springer London, 2008. ISBN 978-1-84800-113-8.

Warren S. McCulloch and Walter Pitts. *A logical calculus of the ideas immanent in nervous activity*, pages 15–27. MIT Press, Cambridge, MA, USA, 1988. ISBN 0-262-01097-6.

Mercedes G. Merayo, Iksoon Hwang, Manuel Núñez, and Ana Cavalli. A statistical approach to test stochastic and probabilistic systems. In *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM '09, pages 186–205, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-10372-8.

C. T. Ng, Natalja M. Matsveichuk, Yuri N. Sotskov, and T. C. Edwin Cheng. Two-machine flow-shop minimum-length scheduling with interval processing times. *Asia-Pacific Journal of Operational Research (APJOR)*, 26(06):715–734, 2009.

James Nutaro. Discrete event simulation of continuous systems. In *Handbook of Dynamic Systems Modeling*, 2005.

James Joseph Nutaro. *Parallel discrete event simulation with application to continuous systems*. PhD thesis, 2003. AAI3119971.

H.A. Oldenkamp. Probabilistic model checking : a comparison of tools, May 2007.

Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 3rd edition, 2008. ISBN 0387789340, 9780387789347.

Diana El Rabih, Gael Gorgo, Nihal Pekergin, and Jean-Marc Vincent. Steady state property verification for very large systems. *International Journal of Critical Computer-Based Systems*, 2011.

Lawrence R. Rabiner. Readings in speech recognition. chapter A tutorial on hidden Markov models and selected applications in speech recognition, pages 267–296. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990. ISBN 1-55860-124-4.

Sheldon M. Ross. *Simulation*. Academic Press, 4 edition, August 2006. ISBN 0125980639.

S. Scalise, H. Ernst, and G. Harles. Measurement and modeling of the land mobile satellite channel at ku-band. *Vehicular Technology, IEEE Transactions on*, 57(2):693 –703, march 2008. ISSN 0018-9545. doi: 10.1109/TVT.2007.906338.

AG. Sciascia, S. Scalise, H. Ernst, and R. Mura. Statistical characterization of the railroad satellite channel at ku-band. *In proceedings of the International Workshop of Cost Actions*, pages 272–280, 2003.

Koushik Sen, Mahesh Viswanathan, and Gul Agha. Statistical model checking of black-box probabilistic systems. In *In 16th conference on Computer Aided Verification (CAV'04), volume 3114 of LNCS*, pages 202–215. Springer, 2004a.

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, 2004b. IEEE Computer Society. ISBN 0-7695-2185-1. doi: 10.1109/QEST.2004.22.

Koushik Sen, Mahesh Viswanathan, and Gul Agha. Vesta: A statistical model-checker and analyzer for probabilistic systems. In *Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, pages 251–, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2427-3. doi: 10.1109/QEST.2005.42.

T. T. Soong. *Fundamentals of Probability and Statistics for Engineers*. Wiley-Interscience, 1 edition, April 2004. ISBN 0470868147.

William J. Stewart. *Probability, Markov Chains, Queues, and Simulation: The Mathematical Basis of Performance Modeling.* Princeton University Press, Princeton, NJ, USA, 2009. ISBN 0691140626, 9780691140629.

Hana Ševčíková, Alan Borning, David Socha, and Wolf-Gideon Bleek. Automated testing of stochastic systems: a statistically grounded approach. In *Proceedings of the 2006 international symposium on Software testing and analysis*, ISSTA '06, pages 215–224, New York, NY, USA, 2006. ACM. ISBN 1-59593-263-1.

Gabriel A. Wainer. *Discrete-Event Modeling and Simulation: a Practitioner's approach.* CRC Press. Taylor and Francis, 2009.

Wei Wei, Bing Wang, and Don Towsley. Continuous-time hidden Markov models for network performance evaluation. *Perform. Eval.*, 49:129–146, September 2002. ISSN 0166-5316.

Håkan L. S. Younes. Probabilistic verification for "black-box" systems. In *CAV*, pages 253–265, 2005.

Håkan L. S. Younes, Edmund M. Clarke, and Paolo Zuliani. Statistical verification of probabilistic properties with unbounded until. In *SBMF*, pages 144–160, 2010.

Hakan Lorens Samir Younes. *Verification and planning for stochastic processes with asynchronous events.* PhD thesis, Pittsburgh, PA, USA, 2004. AAI3159989.

C. S. Yu. Pitman efficiencies of Kolmogorov-Smirnov test. *The Annals of Mathematical Statistics*, 42(5):1595–1605, 1971.

Bernard P. Zeigler, Herbert Praehofer, and Tag G. Kim. *Theory of Modeling and Simulation, Second Edition.* Academic Press, 2 edition, January 2000. ISBN 0127784551.

Armin Zimmermann. *Stochastic Discrete Event Systems: Modeling, Evaluation, Applications.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. ISBN 3540741720.