Two quantitative extensions to perform formal testing of timed systems

Proyecto Fin de Máster
(Master Thesis)

Author: César Andrés Sánchez
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Advisor: Manuel Núñez García

"Siembra un acto y cosecharás un hábito.
Siembra un hábito y cosecharás un carácter.
Siembra un carácter y cosecharás un destino."

Charles Reade (1814-1884)

A mi familia y amigos.

Agradecimientos

Gracias a mi familia, a mi director, y a todos aquellos compañeros y amigos que inestimablemente me han brindado su incondicional apoyo. Sin ellos esto no hubiera sido posible.

Contents

List of Figures

Chapter 1

Introduction

In Computer Science, *formal methods* refer to the area that is concerned with the application of mathematical techniques to the specification, design, implementation and verification of computer hardware and (more usually) software. The use of formal methods in Computer Science is increasing because of the following two main reasons:

The correctness problem producing software that is "correct" is famously difficult but, by using rigorous mathematical techniques, it is possible to make provably correct software.

Programs are mathematical objects and they are expressed in a formal language, they have a formal semantics, and programs can be treated as mathematical theories.

The use of formal methods is especially relevant in reliable systems where, due to safety and security reasons, it is important to ensure that errors are not included during the development process. Formal methods are particularly effective when used early in the development process, at the requirements and specification levels, but can be used for a completely formal development of a system.

The formal representation of real systems allows to rigorously analyze their properties. In particular, it allows to establish the *correctness* of the system with respect to a specification or the fulfillment of a specific set of required conditions, to check the semantic *equivalence* of two systems, to analyze the *preference* of a system to another one with respect to a given criterion, to predict the possibility of *incorrect behaviors*, to establish the *performance* level of a system, etc.

In this way, *formal testing* techniques [BU91a, LY96, Lai02, RMN08, HBB+08] allow to test the correctness of a system with respect to a specification. Formal testing originally

targeted the functional behavior of systems, such as determining whether the tested system can, on the one hand, perform certain actions and, on the other hand, does not perform some unexpected ones. The application of formal testing techniques to check the correctness of a system requires to identify the *critical* aspects of the system, that is, those aspects that will make the difference between correct and incorrect behaviors. While the relevant aspects of some systems only concern *what* they do, in some other systems it is equally relevant *how* they do what they do. In the last years formal testing techniques also deal with non-functional properties. For instance, the probability of an event to happen or the time that it takes to perform a certain action may be considered critical in a real-time system. The activity of conformance testing is essentially focused on verifying the conformity of a given implementation to its specification. In most cases testing is based on the ability of a tester that stimulates the implementation under test and checks the correction of the answers provided by the implementation. The application of formal testing techniques to check the correctness of a system requires to identify the *critical* aspects of the system, that is, those aspects that will make the difference between correct and incorrect behavior. In this line, the time consumed by each operation should be considered critical in a real-time system. The testing community has shown a growing interest in extending these frameworks so that not only functional properties but also quantitative ones could be tested. Thus, during the last years there have been several proposals for timed testing (e.g. [MMM95, CL97, HNTC99, SVD01, EDK02, KT05, HW05, BB05, NR06, MNR08b, MNR08c]). However, in some situations this activity becomes difficult and even impossible to perform. For example, this is the case if the tester is not provided with a direct interface to interact with the implementation under test (`IUT`).

Most formal testing approaches consist in the generation of a set of tests that are applied to the implementation in order to check its correctness with respect to a specification. Thus, testing is based on the ability of a tester to stimulate the `IUT` and check the correction of the answers provided by the implementation. However, in some situations this activity becomes difficult and even impossible to perform. For example, this is the case if the tester is not provided with a direct interface to interact with the `IUT` or the implementation is built from components that are running in their environment and cannot be shutdown or interrupted for a long period of time. The activity of testing could be specially difficult if the tester must check temporal restrictions. In these situations, the instruments of measurement could be not so precise as required or the results could be distorted due to mistakes during the observation. As a result, undiscovered faults may result in failures at runtime, where the system may

perform untested traces. In these situations, there is a particular interest in using other types of testing techniques such as *passive testing*. In passive testing the tester does not need to interact with the IUT. On the contrary, execution traces are observed and analyzed without interfering with the behavior of the IUT. Passive testing has very large application domains. For instance, it can be used as a monitoring technique to detect and report errors (this is the use that we consider in this Master Thesis), in network management to detect configuration problems, fault identification, or resource provisioning, it can be also used to study the feasibility of new features as classes of services, network security, and congestion control. Usually, execution traces of the implementation are compared with the specification to detect faults in the implementation. commonly the specification has the form of a finite state machine (FSM) and the studies consist in verifying that the executed trace is accepted by the FSM specification. A drawback of the first approaches is the low performance of the proposed algorithms (in terms of complexity in the worst case) if non-deterministic specifications are considered. In passive testing we remark an approach proposed in [CGP01]. There, a set of properties, called *invariants*, were extracted from the specification and checked on the traces observed from the implementation to test their correctness. One of the drawbacks of this work was the limitation on the grammar used to express invariants. A new formalism that overcomes this restriction for expressing invariants was presented in [ACN03]. It allows to specify wild-card characters in invariants and to include a set of outputs as termination of the invariant. This approach was extended and revised in [BCNZ05]. There, a new kind of invariants was introduced: Obligation invariants.

The work reported in this Master Thesis extends the approach proposed in [BCNZ05] in order to deal with timed restrictions. Next, we informally introduce the formalism to express temporal conditions in the invariants: Timed invariants. Intuitively, an invariant expresses the fact that each time the implementation under test performs a given sequence of actions, then it must exhibit a behavior in a lapse of time reflected in the invariant. We distinguish between timed restrictions related to each action in the trace represented in the invariant and the one corresponding to the whole trace. For example we could represent properties as

> *"Each time that a user applies "a" and observes "y" the amount of time the system spends to perform the action "y" is between 3 and 5 time units; if after performing some operations the user applies "b" then he must observe "z" before 2 time units and the performance of all these actions must not exceed 10 time units".*
>
> *"Each time that a user asks for connection and the connection is granted, if after performing some operations the user asks for disconnection then he is discon-*

nected, and the amount of time the system spends to perform the disconnection is between 1 and 4 time units."

"Each time that a user asks for a resource (e.g. a web page) either the resource is obtained or an error is produced, it does not matter the amount of time that has passed."

In this work we study two formal testing methodologies where the temporal behavior of systems is taken into account. A simple extension of the classical concept of *Temporal Finite State Machine* (`TFSM`) will allow a specifier to explicitly denote temporal requirements. In one methodology we consider that the notion of time is represented by using fixed timed values and intervals. In the second methodology time is represented by using stochastic time. With these approaches, performance values may change after each transition. In fact it may happen that if we perform two (or more) times the same transition, each performance takes a different amount of time. A transition such as $s \xrightarrow{i/o}_t s'$ indicates that if the machine is in state s and receives the input i, it will perform the output o and reach the state s' after t units of time. A transition as $s \xrightarrow{i/o}_F s'$ indicates that if the machine is in state s and receives the input i, it will perform the output o and reach the state s' after a certain time t according to the probability distribution function F.

In our approach, we perform two types of property verification: One on the specification and another one on the traces generated by the implementation. Due to the fact that we assume that timed invariants can be supplied by the tester, the first step must be to check that the invariant is in fact correct with respect to the specification. Two new extensions of the algorithm proposed in [BCNZ05] to check this correctness are provided, taking into account the timed conditions that appear in timed invariants. The next step is to check whether the trace produced by the `IUT` respects timed invariants. In this case, we propose new algorithms that are adaption of the classical algorithms for string matching. They work, in the worst case, in time $O(m \cdot n)$ where m and n are the length of the trace and the invariant, respectively. Let us remark that we cannot achieve complexities as good as the ones in classical algorithms because we have to find all the occurrences of the pattern.

The rest of this Master Thesis is structured as follows. In Chapters 2 and 3 we review the state-of-the-art on formal testing of timed systems and on passive testing and monitoring techniques, respectively. In Chapter 4 we present two new methodologies to perform passive testing based on invariants for systems that present temporal restrictions. In both frameworks, we present algorithms to decide the correctness of the proposed invariants with respect to a given specification. Once we know that an invariant is correct, we check whether

the execution traces observed from the implementation respect the invariant. In Chapter 5, we present the main features of a new tool, called PasTe, that helps in the automation of our passive testing approach. In particular, all of the algorithms presented in this work are fully implemented. Finally, in Chapter 6 we give our conclusions and sketch some lines for future work.

Some of the contents of this Master Thesis have already been published in the ATVA 2008 Conference [AMN08].

Chapter 2

State-of-the-Art: Testing and Timed Extensions

With the growing significance of computer systems, techniques that assist in the production of reliable software are becoming increasingly important. The complexity of many computer systems requires the application of a battery of such techniques. Two of the most promising approaches are *formal methods* and *testing*. Traditionally formal methods and testing have been seen as rivals. Thus, there was very little interaction between the two communities. In recent years, however, these approaches are seen as complementary [Hoa96, BBC+02]. The links between testing and formal methods go well beyond generating tests from a formal specification. The presence of a formal specification or model makes it possible for the tester to better understand what it means for a system to pass a test. This may be achieved through the use of *test hypotheses* [Gau95] or *design for test conditions* [IH97, HI98]. Similar ideas can be found in the generation of tests from finite state machines in which some fault model is assumed (see, for example, [IT97]). Using these approaches it is possible to generate tests that determine correctness under certain well understood conditions, circumventing Dijkstra's famous aphorism that testing can show the presence of bugs, but never their absence [Dij72].

As we observe, since there is plenty of work in the area of formal testing methodologies, we have reviewed the field by concentrating on general ideas, without paying too much attention to specific papers on the area. In this chapter we will restrict the scope to deal only with formal testing techniques for analyzing the *temporal* behavior of systems. First, we will briefly consider some formalisms that have been proposed for representing real-time systems and systems where time plays a relevant role in terms of temporal constraints, performance,

etc. Next, we will describe techniques that have been proposed to test the correctness of implementations with respect to specifications defined by means of the previous formalisms.

2.1 Formalisms to represent Time

Though time affects any system, the interest of researchers to explicitly include it in formal models is relatively recent. The development of networked and distributed systems, multimedia systems, and real-time systems led to a new scenario where performance aspects and temporal requirements became relevant topics of interest. Essentially, the behavior of a system may depend on the passage of time in the following two ways:

- After a user request a computation, the system may need a perceptible time to complete it. This amount of time may make the difference between an acceptable and an unacceptable performance, that is, between a system that is considered either correct or incorrect.

- Sometimes the passage of time is not a drawback but a requirement. That is, a system may be required to wait a given amount of time before a certain operation is performed, while performing it before this time could be incorrect.

Explicitly including this information in specification formalisms is not straightforward. On the one hand, the syntax of the language must allow the representation of these requirements in an expressive way. On the other hand, the operational semantics must be able to denote the passing of time in a handleable fashion. For instance, if a system may idle 3 seconds then it can also stay idle 2 seconds. In general, given a system configuration, the different time amounts that are relevant for the description of the system represent a very high number (they could be even uncountable!). Let us note that, even if we consider that time is discrete, making the language semantics to denote arbitrary delays by successively iterating the pass of a given atomic *unit* of time is unfeasible. Hence, a suitable representation of arbitrary delays is required. Similarly, if the specification allows to wait any amount of time between 4 and 12, then it is unfeasible to make the specification to *explicitly* denote all the allowed time values one after another (i.e., 4, 5, ..., 12 if time is discrete). Instead, a proper compact representation is required.

Moreover, there are several ways to interpret temporal requirements. On the one hand, we may consider that temporal requirements are *strict*. For example, we may require that some event occurs after exactly 3 seconds. On the other hand, we may relax this requirement

by considering temporal *intervals*. Moreover, we may consider that temporal requirements are defined in *probabilistic terms*, that is, we require that something occurs before t units of time with a given probability p. These differences affect both the definition of specifications and the behaviors extracted from them.

Formalisms to describe timed systems are usually extensions or adaptations of other formalisms previously proposed to denote systems where time is not explicitly considered. Next we briefly sketch some of these formalisms. First, we will consider formalisms where time is not defined in probabilistic terms. This does not necessarily mean that we have to express time requirements by using fix values. As we will show later, specifications may denote that *any* time that fulfills a given constraint (e.g., belonging to a given interval) is allowed. Next, systems allowing to denote *stochastic time* will be reviewed.

2.1.1 Representation of non-probabilistic time

Next we consider specification formalisms where time is not defined in probabilistic terms. Several models for the analysis and specification of timed systems have been devised, including timed Petri nets [Sif77, Zub80], the duration calculus [CHR91], and a variety of extensions of automata with time information (e.g., [LV96, SGSL98, VPC02, LSV03, PCVM04, BVC05]). However, the model that has been widely accepted is *timed automata* [AD90, AD94, HNSY94]). Inherently, if we wish to represent *dense* time then a timed system induces an infinite behavior. Timed automata propose a symbolic way to describe this infinite behavior. In so doing, they enable a full state space exploration by traversing a *symbolically finite* state space. This is mainly the reason for the popularity of this model.

A timed automata represents the behavior of a system in terms of a fixed set of clocks. During the execution of the automaton, each clock has an associated time value. When the automaton sojourns in a state, clock time values increase in a synchronous manner. The transitions of the automaton are instantaneously executed, may depend on some condition on clocks, and may cause some clock to be set to a given value. More precisely, timed automata transitions are of three kinds:

- *Actions*, used to express the occurrence of events and to synchronize system components (when composing in parallel several timed automata),
- *guards*, expressing conditions on clocks, and
- *clock setting* events.

Let us note that the representation of time passage in a timed automata is *symbolic* in the sense that the temporal behavior of the system is expressed by means of events like clock setting and clock constraints instead of *concrete* (real-valued) time transitions. This makes it possible have a finite representation of the system behavior, hence the chance of analyzing some of its properties in a computable way, even if the time domain is assumed to be (continuously) infinite. On the other hand, the semantics of timed automata (the meaning of the symbolic representation) is usually defined in terms of an inherently infinite *concrete* semantic model which makes use of real-valued timed transitions.

Timed automata have served as a model for many model checking algorithms, see for example [ACD93, HNSY94, YPD95, ACH94, DOY95, DY95], and tools that implement these algorithms such as Kronos ([DOTY96, BDM+98]), UPPAAL ([BLL+96, LPY97, BLL+98]) and HyTech ([HHWT95]). These tools were used in a diversity of case studies (e.g., [HWT95, DY95, BGK+96, MY96, DKRT97, LPY98]).

A process algebraic approach for timed systems has also been pursued. Originally, time was included in a discrete fashion (e.g. [MT90, NS91, Gro91, Han91, QFA93, BB96]), that is, time is represented as a *tick* action that describes the passage of a single time unit. These process algebras are adequate to model digital systems but it could be argued that they cannot describe real-time systems in a natural way.[1] For this reason, dense time process algebras were developed (e.g., [Yi90, BB91, SDJ+92, LL97]). Some of these process algebras have been formally related to timed automata (e.g., [NSY92, BHKR95]). However, these relations only provide a semantic connection and none of them is complete in the sense that such a connection is bijective. Languages that completely represent timed automata have been also defined in [AH94, YPD95, LV96].

2.1.2 Representation of stochastic time

The representation of time in probabilistic terms has also been considered in the literature. The analysis of stochastic systems has received a lot of attention but, traditionally,

[1] The topic of the (un-)suitability of discrete time to denote timed systems (vs dense time) has typically yielded a big controversy. On the one hand, the time in electronic and computational timed systems uses to be digital indeed. Since no human-designed system will ever be able to have infinite temporal precision, we may argue that we cannot design/construct systems in such a way that dense time is actually taken into account. Moreover, no observation or testing device has such a precision. On the other hand, supposing the existence of a minimal time tick may lead to the wrong assumption that analysis techniques can be based on systematically considering *all* available times. Let us note that the amount of times to be considered is astronomic in general. A way to avoid the temptation to make such an assumption is using dense time in models.

outside the community of formal methods. First, mathematicians defined models for the analysis of stochastic processes. These models, which include the so-called *continuous time Markov chains*, in the following CTMC, were used to analyze the performance of systems. But systems started to become more complex and there was a need for a more sophisticated notation to specify performance models. This led to the definitions of models such as *queueing networks* [Kle75, HP92] and a diversity of *stochastic Petri nets* [ACB84, ABC+95].

Models based on CTMCs represent the timing of events as stochastic variables following an exponential distribution. This restriction is the key for the vast analytical and numerical theory that supports CTMCs. Restricting to exponentially distributed durations gives the advantage of having activities for which the *memoryless* property holds. This property basically says that at each point of time in which an activity has started but not terminated yet, the residual duration of the activity is still distributed as the entire duration of the activity. Such a property makes it possible to represent *timed* behavior of systems by using a CTMC, that is, a simple continuous process where in each time point the future behavior of the process is completely independent of its past behavior and depends on its current state only (Markov property). In fact, the memoryless nature of time in the Markovian approach makes it possible to avoid the explicit representation of time passage in system descriptions. For instance, consider a simple example of two exponentially timed activities with rates (the parameters of the exponential distribution) λ and μ executed in parallel. The resulting CTMC is the one in Figure 2.1. Transitions in a CTMC represent exponentially distributed delays and choices in a state of a CTMC are resolved via a *race policy*, that is, the delays represented by the outgoing transitions are executed in parallel and the first delay that terminates determines the transition to be performed. Therefore, in the example, both delays synchronously count from the initial state and when one of them terminates, the corresponding transition is executed. Such a transition leads to a state where the other delay counts its residual duration until it terminates as well. Note that, because of the memoryless property, the residual duration of the delay is also exponentially distributed and with the same rate. Hence the CTMC of Figure 2.1 is an adequate representation of the parallel execution of the two considered activities.

Unfortunately, restricting to exponential distributions is often not realistic and may lead to results that are not sufficiently accurate. For instance, in the analysis of high-speed communication systems or multi-media applications the correlation between successive packet arrivals tends to have a constant length; therefore, the usual Poisson arrivals and exponential packet lengths are no longer valid assumptions [BKLL95]. More general models, such as *gen-*

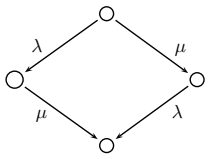


Figure 2.1: Parallel of exponential delays: $\lambda \| \mu$.

eralised semi-Markov processes ([Whi80, Gly89, Cas93, She93]), allow for the description of timings that may depend on any distribution. Unfortunately, none of these models provides a suitable framework for the composition of distributed and communicating systems.

The formal methods community tackled the description and analysis of stochastic systems by means of *probabilistic process algebras* (e.g., [GJS90, Han91, GSS95, BBS95, NFL95, Low95, NF95, CCVP01, Núñ03, CCV+03]), although they were initially intended for verification purposes rather than performance analysis. In addition, they only deal with discrete probability distributions, and most of them do not include a notion of time. It is the seminal work of Ulrich Herzog [Her90] were process algebras and performance models were combined. Stochastic process algebras were thus introduced. Taking advantage of the analytical framework provided by continuous time Markov chains, so-called Markovian process algebras were devised. They include TIPP [HR94], PEPA [Hil96], EMPA [BG98], MPA [Buc94], and IMC [Her98]. The general approach, including any continuous distribution function, has also been addressed (e.g., [GHR93, HS95, BKLL95, Pri96, BBG98, LN00, BG02, LNR04, DK05]). Each approach deals with parallel composition in a different manner. When no interaction is involved, parallel composition can be easily defined in Markovian process algebras. If arbitrary distributions are allowed, this definition proved to be more complicated yielding complex or infinite semantic objects. A comparison of, on the one hand, models restricted to exponential distributions and, on the other hand, models allowing general distributions is provided in [BD04].

Synchronization leads to more complicated decisions. In fact, not all the proposed stochastic process algebras provide a symmetric interaction. For instance, EMPA requires that at most one of the synchronizing components is time dependent while the others must be *passive*. In [Hil94] synchronization takes place by means of a *patient* communication, that is, it takes place when all the components that intend to synchronize are ready to do it. This approach is adopted also by [HS95, BKLL95, Her98]. The process algebra PEPA also uses this approach but needs to approximate the distribution of time in order to stay in the domain of exponential distributions. Let us note that patient communication can

model the aforementioned asymmetric synchronization: A passive process is always ready to synchronize. Another issue that arises as a consequence of considering parallel composition in interleaving models (such as automata-based models) is non-determinism.

2.2 Testing temporal systems

In this section we present the most relevant approaches to test timed systems that can be found in the literature. Let us note that time does exist in any system, regardless of whether we explicitly represent it in a model. Actually, we may consider that a non-timed system is a particular case of timed system where the future behavior at any point depends only on the past *sequence* of inputs and outputs (i.e., time never induces a change of state) and the only way to assess the system performance is to count the *number* of operations performed before a result is provided.

Testing a *non-timed* system is not an easy task. In general, there are infinite ways to interact with a system. Hence, the correctness of a `IUT` can only be claimed after we check that it answers the expected value for *all* of them. If the interaction with the `IUT` consists in successively providing inputs and observing outputs, then new available interaction cases are found by simply considering longer sequences of inputs and outputs. Let us note that the time we can spend testing a `IUT` is finite. Thus, the amount of tests to be applied (as well as the size of each of them) must be finite as well. As a result, the difficulty to testing systems lies on the impossibility to make tests to reach and check *any* future point in the `IUT`. Still, we can partially test the `IUT` up to a given future point. For instance, we can consider a set of tests where *any* sequence of less that n inputs is proposed and we can apply it to the `IUT`.

Unfortunately, the previous considerations might not be true for timed systems. If time does affect the behavior of system, then tests do not only consist in the sequence of inputs that is applied to the `IUT` but also have to take into account *when* they will be applied. Hence, the number of choices to define a test dramatically increases. On the one hand, if time is assumed to be discrete then all time values allowed by the specification must be considered, potentially leading to an astronomical number of choices. In particular, if the specification allows to perform an operation from some time t *on*, then infinite choices are available (t, $t+1$, $t+2$, ...). On the other hand, if time is dense then the number of choices could be uncountable! Hence, composing and applying a test suite including any available interaction with the `IUT` up to a given future point (i.e., up to a given time) could be unfeasible or simply impossible.

There are different ways to tackle the problem of testing timed systems. First, we may accept the impossibility to cover all behaviors of the IUT and test only some of them that are considered specially relevant or somehow representative of the rest of behaviors that will not be checked. An alternative way consists in adding some *hypotheses* about the behavior of the IUT such that the application of a finite test suite may be enough to guarantee the correctness of the IUT. In any case, being able to define the test suite to be applied to completely check the correctness of the IUT in the general case is desirable. Though it is unfeasible in general to completely construct and apply such a suite, having a constructive way to find tests in this set is useful to define techniques allowing to find *relevant* tests according to a given criterium. Most proposals in the literature focus on this task: Finding complete test suites as a first step in the development of a method where *only* some tests in them are applied.

As far as we are concerned, almost all formal testing methods for timed systems concern non-probabilistic time. Only in [NR03] a method to test stochastic systems is presented. The method is *probabilistic* due to the nature of the analysis. Since a black box approach is assumed, the interaction with the IUT by providing inputs and receiving outputs does not allow the tester to *read* the random variables defining the IUT behavior. Instead, concrete times are obtained in each observation. So, all that can be done is to (statistically) *extract* these random variables from the IUT by repeating the same interaction several times and collecting a sample. Then, a *hypothesis contrast* is applied to check whether it is feasible that such an IUT sample is produced by a random source behaving as the corresponding *specification* random variable says. Since hypothesis contrasts provide diagnostics up to a given confidence levels, the correctness of the IUT is assessed up to these levels as well. Let us remark that in the context of testing *a la* de Nicola & Hennessy, the work on stochastic processes is also very limited, being [BC00, LN01] the only two proposals, for Markovian and generally distributed stochastic processes, respectively.

[BB04] presents a temporal extension of the ioco theory [Tre96, Tre99, Tre08]. As in that case, the concept of *quiescence* (a quiescent state is a state where the system cannot produce outputs) is introduced in models as a way to increase the distinguishing power of tests in the testing theory. Obviously, this requires to assume that tests can detect it. Basically, it is assumed that we are provided with a bound that is the explicit representation of the time a system should idle until quiescence can be concluded. Treating quiescence as a special sort of system output provides with information to differentiate systems that have intuitively different deadlocking properties (e.g., [Lan90, FNQ95, Tre08]). The use of quiescence gives rise to a family of implementation relations parameterized by observation

durations for quiescence. In this framework, timed input-output labelled transition systems are used to define specifications. A sound and complete test derivation algorithm is presented. Similar to [Tre08], the algorithm is non-deterministic, representing each choice a different path in the specification. By considering all nondeterministic choices, a test suite is constructed. Let us note that, as in the non-timed approach, the set constructed by the algorithm is infinite in general.

In [BB05] an extension of the previous work, allowing to deal with multiple channels, is considered. A model of timed multi input-output transition systems is presented. It allows to model timed systems that communicate with the environment via multiple input and output channels. Formally, channels are represented as a partitioning of the sets of input and output actions, each partition class defining the inputs (outputs) belonging to an individual input (output) channel. Besides, the input enabling assumption (that is, that all inputs are enabled anytime) is relaxed by allowing some input sets to be enabled while others remain disabled. Moreover, the general bound used in timed systems to detect quiescence is also relaxed, and different bounds for different sets of outputs are allowed. By considering the theory presented in the previous work, an alternative theory for timed testing with repetitive quiescence, and allowing the partition of input sets and output sets, is introduced. A new conformance relation parameterized by these factors is proposed. Besides, a parameterized test derivation procedure (again nondeterministic) is developed and shown to be sound and complete with respect to a new conformance relation. Again, test suites provided by the procedure are infinite in general.

Other theories for testing reactive systems providing complete test suites [MMM95, PS97] require infinite test suites to be complete as well. So, in these methodologies the outcome of testing a finite number of cases only approximates a complete test. As we have already pointed out, only finite test suites can be applied in practice. Hence, we may consider such approaches as theoretical roots to build other theories where finite test suites are constructed and applied. Next we consider some methods to produce finite test suites. In this case, we have two possibilities. First, a finite test suite may be *complete* to test the `IUT` with respect to the specification. This property is usually met by adding some strong hypotheses about the `IUT` or the environment. Second, the test suite may be uncomplete. In this case, methods to find tests with a good capability to find errors in the `IUT` are pursued.

In [SVD01] a generalization of the classical testing theory for Mealy machines to a setting of dense real-time systems is presented. A model of *timed I/O automata* is introduced, inspired by the timed automata model of [AD90, AD94], together with a notation of test

sequence for this model. The main contribution of this work is a test generation algorithm for black-box conformance testing of timed I/O automata. Although it is highly exponential and cannot be claimed to be of practical value, it is the first algorithm that yields a *finite* and complete set of tests for dense real-time systems. Apart from supporting the automatic generation of timed tests, the model allows a loose coupling of inputs and outputs, unlike the usual Mealy machine style where inputs and outputs occur paired in a single transition.

The test generation algorithm for this model is provided in the style of well-known finite state machine based methods that were commented in the previous chapter. The main problem involved is that in general the state space of a timed automaton is (uncountably) infinite. To obtain a finite set of tests, a discretization of the state space is required which is still sufficiently refined to detect all possible errors.

The paper proposes the first algorithm that (albeit under some *strong* assumptions) yields a finite and complete set of tests for (dense) real-time systems. Only timed automata where outputs are isolated and urgent are considered. The first condition states that, at any given state, the automaton can only output a single action. The second condition states that, at any given state, if an output is possible, then time cannot elapse. This essentially means that outputs must be emitted at precise points in time. Even though the algorithm itself is highly exponential, the presented concepts and techniques have actually served for subsequent more practical algorithms. Some optimizations are sketched in the paper. Besides, the approach also tries to support incomplete but practically useful methods for testing timed systems as in [MMM95, CL97]. The conformance relation applied in this work is bisimulation, though it reduces to trace equivalence because determinism is assumed.

In [FPS01] another technique to testing real-time systems through the derivation of executable test cases on a specification, modelled as a timed automaton, is presented. The main peculiarity of this work is how the authors try to make the testing technique to be *feasible*. While other studies focus on reducing the specification formalism in order to be able to derive test cases feasible in practice, in this work test cases are derived from specific *test purposes* given by the user. These test purposes express specific user properties. In addition, though timed automata are used to define specifications, the study deals with an equivalent representation of timed automata: *Clock region graphs*. Clock region graphs are extracted from the timed automata by considering all the possible valuations of clocks that are equivalent in terms of fulfilling (or not) the requirements imposed by the automata in each guard. In particular, a clock region is an equivalence class induced by this equivalence relation. Then, a clock region graph is a graph where the states are induced by these regions.

A test purpose is modelled by an acyclic graph: All paths of this graph which are found on the specification will be considered as a test case. The timing constraints of both the specification and the test purpose should be the same. The application of tests allows both to discriminate between the different cases (input, output or waiting) and to test a *representative* part of the infinite set of clock values. A simple example illustrates the use of this technique.

In [CG98] a formal method for generating conformance tests for real-time systems is presented. The algorithm is complete in that, under the proposed set of test hypotheses, if the system being tested passes every test generated then the tested system is *bisimilar* to its specification. The algorithm provides *finite* test suites. Because it has exponential worst case complexity and finite state automata models of real-time systems are typically very large, the authors accept that a judicious choice of model is critical for the successful testing of real-time system. In contrast to [SVD01], where the authors acknowledge that their method is not practical due to its exponential complexity but no way to tackle this problem is proposed, the test suite provided by the test derivation algorithm of [CG98] can be further manipulated to find redundancies. In this way, the actual number of tests to be applied is reduced and practical test suites can be constructed while the completeness of the analysis remains. This claim is shown in the paper with several examples. Unfortunately, though these ideas are presented as a *methodology*, no automatic procedure is provided for making such a reduction in the general case.

The proposed temporal model is based on rules of the form "If G then A between L and U," where G is a guard on the variables and clocks, A is an action on them, and L and U are the lower and upper bounds of the time spent in this transition. The completeness of the test suites provided by the test derivation algorithm is met by adding a high amount of test hypotheses. In particular, it is required that if two systems are non-equivalent then their behavior differs in at least one unit of time, and that there are neither livelocks nor deadlocks. Moreover, the determinism of the `IUT` is assumed. Other more standard hypotheses assume that the proposed formalism is adequate for defining the `IUT` and that the `IUT` can be reset at any time. In [Car99] the previous work is moved to the context of timed automata. In particular, the language notation used in UPPAAL to represent timed automata is adopted. Existing test generation methods for I/O automata are adapted for the domain of timed automata. The method addresses both the problem of *untestable* timed automata computations and that of too many possible tests, and tries to provide a practical method for managing conformance testing of real-time systems. In this regard, the author

recalls again that, though the method in [SVD01] refers to a very general class of timed automata, it generates an astronomically large number of test sequences. It is shown how to deal with timed automata computations for *testing purposes* using digitization, hiding, and input-output labelling. However, with respect to other related work, the main contribution consists in showing how structured test management methods from mainstream software engineering can be applied in the real-time context. Based on this idea, a procedure is proposed where each test purpose is represented by a timed automata specification, sometimes a modification of the original specification, together with a list of visible and invisible variables. The model which is generated from such a view (called *test view*) is expected to be simpler than the model of the original specification and should generate a feasible number of tests. Basically, the proposed method consists in splitting the specification into different test purposes. Then, a different view, that is a new version of the specification that only concerns a specific aspect of the system, is constructed for each test purpose. In this process, the system variables that are *visible* and those that are *hidden* for each test purpose are chosen. Tests are extracted for each view by using an adaptation of [Cho78] and applied to the implementation. It is shown that the test suite constructed for each test purpose is complete to check the conformance of the `IUT` with respect to *this* purpose. By using this partial testing approach, testing each test purpose is simpler because the explosion of tests that is due to the necessity to explore all interactions of different parts of a system to each other is partially reduced. Let us note that each of these parts is not a component but rather a functionality or usage mode.

Regarding the use of timed automata in the testing framework, the author argues that standard semantics for timed automata usually present several problems for testing. First, timed automata have a dense time model and so their traces include behavior which cannot be observed in an experiment. For example, events may be specified to occur at different, but arbitrarily close times, leading to different outcomes. However, to an observer the ordering of two arbitrarily close events cannot be distinguished, and a tester does not have sufficient control over a physical system to offer inputs at arbitrarily close, but different times. According to the author, a more appropriate model for observing real-time systems is a *digital clock approximation* [HMP92]. Secondly, because UPPAAL timed automata differ from I/O automata in having persistent data variables, clocks, and closed world specifications, assumptions underlying test generation methods for I/O automata need revising for timed automata. In particular, for real-time systems new assumptions regarding state identification, input enabling, and extra implementation states are introduced in this work.

In [Car00] the previous work is refined by introducing an intermediate *testable* language in the process. UPPAAL timed automata are transformed into *testable timed transition systems* (TTTSs) using a *test view*. Fault hypotheses and a test generation algorithm which extracts test cases from TTTSs are defined. Test suites constructed by the algorithm are complete with respect to the chosen test view. Besides, the management of persistent data variables allowed by UPPAAL is now supported. Results of applying the method are presented.

In [CL97, CKL97] a framework for testing timing constraints of real-time systems is presented. Test cases are derived from specifications described in the form of a constraint graph, and only the minimum and the maximum allowable delays between input/output events are considered. Though usual testing concepts appear in this paper, tests are proposed to be applied to a *model* rather than to a `IUT`. In particular, the testing approach is proposed as a method to find properties in a system model, that is, *validation* is the main goal of the work. Tests are automatically derived from specifications of minimum and maximum allowable delays between input/output events in the execution of a system. Contrarily to previously commented proposals, time constraints are defined with a different formalism to that used to define specifications. In particular, the test derivation scheme uses a graphical specification formalism for timing constraints, and the real-time process algebra ACSR [BLG93, BL97] for representing tests and process models. ASCR is a timed process algebra based on the synchronization model of CCS that includes features for representing synchronization, time, temporal scopes, resources, requirements, and priorities. According to the authors, the use of an expressive language provided with precise semantics to describe test sequences, like ACSR, has two main advantages. First, tests can be applied to an ACSR model of the software system within the ACSR semantic framework for model validation purposes. Second, ACSR has concise notation and a precise semantics that facilitate the translation of real-time tests into a software test language for software validation purposes.

The authors propose their testing-oriented method of validation as a way to analyze very complex systems. In fact, since the number of tests is chosen by the tester, the method can be used to validate a design specification which has too many states for exhaustive state space exploration based analysis. If tests are carefully chosen according to some criteria then the result of their application may be adequate. A derivation method to automatically extract tests from a model is proposed. Given a test coverage criterium, the algorithm considers all the tests necessary to completely achieving this criterion (this test suite may be still too big), and then choosing the tests that seem to be more *representative* in this set. As an illustration of the method, a case study of using the automatic derivation of tests from

timing specifications for the analysis of a system is presented.

In [HLN+03] real-time conformance test cases are automatically generated from timed automata specifications. This work focuses on showing how to efficiently generate real-time test cases with optimal *execution time*, that is, test cases that are the fastest possible to execute. The proposed technique allows time optimal test cases to be generated either manually, by using formulated test purposes, or automatically, from various coverage criteria of the model. In order to justify their approach, the authors hypothesize that, in the context of testing real-time systems, the fastest test case that drives the specification to some state also has a high likelihood of detecting errors, because this is a stressful situation for the specification to handle. Moreover, they claim that time optimal test suites are interesting for several reasons. First, reducing the total execution time of a test suite allows more behavior to be tested in the (limited) time allocated to testing. Second, it is generally desirable that regression testing can be executed as quickly as possible to improve the turn around time between software revisions. Third, it is essential for product instance testing that a thorough test can be performed without testing becoming the bottleneck, that is, the test suite can be applied to all products coming of an assembly line.

The authors claim that an important problem in generating real-time test cases is to compute when to stimulate the system and expect response, and to compute the associated correct verdict. This usually requires (symbolic) analysis of the model which in turn may lead to the state explosion problem. Another problem is how to select a very limited set of test cases to be executed from the extreme large number (usually infinitely many) of potential ones. The authors propose their method to generate time-optimal test cases and test suites as a way to address both issues.

Models are specified by using a deterministic and output urgent class of UPPAAL style timed automata. The fastest diagnostic trace facility of UPPAAL is used to generate time optimal test sequences. The conformance relation applied in the framework is trace inclusion. Let us note that the method proposed in the paper is based on existing efficient and well proven symbolic analysis techniques of timed automata. Its main contribution is that it addresses time optimal testing as well as coverage criteria for it. Actually, most other work on optimizing test suites focuses on minimizing the *length* of the test suite which is not directly linked to the execution time because some events take longer to produce or real-time constraints are ignored.

In [KT04] a framework for black-box conformance testing of real-time systems is presented. Specifications are modelled as timed automata though, contrarily to other previous

proposals in the literature, nondeterministic and partially-observable timed automata are allowed. A conformance relation, called *timed input-output conformance* or `tioco`, which is a timed extension of the classical `ioco` relation, is proposed. According to `ioco`, A conforms to B if for each observable behavior specified in B, the possible outputs of A after this behavior is a subset of the possible outputs of B. The `tioco` relation is simply defined by including time delays in the set of observable outputs. This permits to capture the fact that an implementation producing an output too early or too late (or never, whereas it should) is non-conforming. The authors compare this relation with other previously considered relations (bisimulation in [CG98], must/may preorder in [HNTC99, NS01], trace inclusion in [HLN+03, KJM03], and trace equivalence in [SVD01]) and argue that it is better suited for testing than the other ones because it leaves more design freedom to potential implementations. Algorithms are proposed to generate two types of tests for this setting: *Analog-clock* tests, which measure dense time precisely, and *digital-clock* tests, which measure time with a periodic clock. A heuristic to generate a test suite that covers all specification edges is briefly discussed. A prototype tool and a small case study are reported.

The aim of the authors is to overcome some limitations of previous methodologies in two directions. First, other work restricts the kind of specifications that can be defined. For example, [SVD01, HLN+03] consider timed automata where outputs are isolated and urgent. Due to the first condition, a specification such as "when input a is received, produce either output b or output c" cannot be expressed in this model. Due to the second, a specification such as "when input a is received, output b must be emitted within at most 10 time units" cannot be expressed. Other works use to consider deterministic or determinizable subclasses of timed automata. For instance, [NS01] uses *event-recording automata* [AFH94] while [KJM03] use a determinizable timed automata model with restricted clock resets. It is also typically assumed that specifications are fully-observable, meaning that all events can be observed by the tester. On the contrary, [KT04] allows to represent non-deterministic and partially observable specifications. In particular, the issue of determinizing tests is addressed *on-the-fly* during test generation and execution.

The second limitation concerns *implementability* of tests. In the typical controversy of dense versus discrete time, the authors claim that, in practice, only digital-time tests can be constructed and applied. However, only analog-clock tests are considered in previuos work. These are tests which can observe the time of inputs precisely and can also react by emitting outputs in precise points of time. For example, a test like "emit the output a at time 1; if at time 5 the input b is received, announce `PASS` and stop; otherwise, announce

FAIL" is an analog-clock test. Unfortunately, analog-clock tests are problematic since they are difficult, if not impossible, to implement with finite-precision clocks. The tester which implements the test of the example above must be able to emit a precisely at time 1 and check whether b occurred precisely at time 5. However, the tester will typically sample its inputs periodically, say, every 0.1 time units, thus, it cannot distinguish between b arriving anywhere in the interval $(4.9, 5.1)$.

In [KT05] the previous work is extended with some new features. On the one hand, a method is provided allowing the tester to define assumptions about the environment of the IUT. Basically, the tester's assumptions about the environment are defined by means of a *timed automata* defining its behavior. This automata is composed with the timed automata defining the actual specification requirements. On the other hand, additional timed automata can also be used to define the interface between the tests and the IUT. For example, they may be used to define a delay between the time each test stimulus is produced and the reception of the signal in the IUT. In addition, some test derivation algorithms producing test suites with respect to different coverage criteria (*state*, *location*, or *edge*) are provided.

In [HNTC99] a timed I/O automaton model, different to standard timed automata, is proposed to specify real-time protocols. This is a *Timed Input Output Automata* extended with *data*. In order to derive test cases from this model, automata are transformed in a kind of I/O Finite State Machine so that classical test generation techniques can be applied. A conformance testing method for this model is proposed. The authors criticize classical timed automata [AD90, AD94] because they do not deal with data values used in communication protocols. For example, sometimes one may want to specify different timeout intervals depending on the size of data to be transmitted. Hence, models that can treat not only time but also data values are needed. It is also desirable that such models have efficient verification and/or testing methods, as timed automata have. Thus, a combination of timed automata with EFSMs is proposed. In testing EFSMs or real-time systems, a given test sequence is not always executable. In order to execute the test sequence, some appropriate input values or execution timing which satisfy its transition conditions must be found. In contrast to the case of EFSMs, in real-time systems the tester can designate the input timing. However, in general, the output timing is not controlled by the tester and it is decided by each IUT itself. Moreover, the executable timing of some action may depend on the execution time of its preceding actions. It is desirable that whenever the preceding output actions are executed, there always exists some adequate input timing such that its succeeding sequence is executable. In fact, the executable timing of each input action in a test sequence can be

specified by a function of the execution time of the preceding actions. The authors propose an algorithm to decide whether a given test sequence is executable. Besides, they propose a method to derive such a function from an executable test sequence automatically using a technique for solving linear programming problems, as well as a conformance testing method using those algorithms. In this model, determinism is assumed, but not output urgency.

In the timed I/O automaton model, each transition corresponds either to an input action or to an output action. In order to describe timing constraints among actions, some variables and one special global time variable, which always holds the current time, are introduced. The variables can hold not only time values but also values expressed as linear expressions of the time values and input data. Each transition condition can be specified by a logical conjunction of linear inequalities of those two types of variables. The conformance relation considered is a must/may testing criterion. To distinguish sequences that can always be executed to completion independent on output timing and sequences that may be executed to completion, may- and must-traceability of transition sequences are defined. A *must-traceable* test sequence can be always executed if some appropriate input timing for its input actions are specified, no matter when its output actions are executed. A *may-traceable* test sequence can be executed only when the execution time of its output actions belongs to the sub-ranges which make the succeeding actions executable.

In addition, the authors present an algorithm for checking the must/may-traceability of given test sequences and obtaining the upper and lower bounds for each input action as functions of the execution time of its preceding actions. As in [Car00, SVD01], test sequences are generated by using checking sequence techniques, but different structures and state verification methods are used. Based on the UIOv-method [VCI90], a conformance testing method for the model is proposed. This method is applied to a Finite State Machine derived from the automaton by simply removing the clock conditions on transitions. The sequences are then checked for their may- and must traceability, and the procedure is re-iterated when necessary. Let us note that this procedure may result in many iterations and in incomplete test suites.

Next we briefly comment other related work providing testing techniques for temporal systems.

- [MNR08c] proposes a formal methodology to test both the functional and temporal behaviors in systems where temporal aspects are critical. This study extends the classical `FSM` model with features to represent timed systems. This formalism allows three different ways to express the timing requirements of systems by using fix time

values, by using random variables, or by considering time intervals. In this work different implementation relations are presented and related, depending on both the interpretation of time and on the non-determinism appearing in systems. Complete studies about how test cases are defined and applied to implementations, and about test derivation algorithms, producing sound and complete test suites, are also presented.

- In [MNR06, MNR08b] the authors introduce a timed extension of the `EFSM`s model. On the one hand, this study considers that (output) actions take time to be performed. This time may depend on several factors such as the value of variables. On the other hand, the formalism proposed in this approach allows to specify timeouts. In addition, the authors develop a testing theory. They defined ten timed conformance relations and relate them. Besides, the papers introduce a notion of timed test and define how to apply tests to implementations. Finally, an algorithm to derive sound and complete test suites, with respect to the implementation relations presented in this approach, is given.

- [EDKE98, EDK02] present an adaptation of the Wp-method [FBK+91] to timed systems. As in previously commented work [Car00, SVD01], test sequences are generated from a timed automata by applying variations of finite state machine checking sequence techniques to a discretization of the state space. Consequently, this approach also suffers from the state explosion problem and produces large number of test sequences.

- [NS01] proposes a fully automatic method for generation of real-time test sequences from a restricted subclass of dense time automata (*event-recording automata* [AFH94]) which restricts how clocks are reset. This approach is based on de Nicola & Hennessy testing theory. A selection technique of timed tests is presented. This technique is based on symbolic analysis and coverage of a coarse equivalence class partitioning of the state space. The proposed conformance relation is a must/may preorder relation.

- [Kho02] assumes a restricted timed automata model where all transitions with the same observable action reset the same set of clocks. The timed automaton is first translated into a (larger) alternative automaton where clock constraints are represented as set-timer and expire-timer events. Based on this, the generalized Wp-method is used to compute checking sequences. Output urgency is not required, but determinism is assumed.

- [CKL98] presents a different approach to test generation and selection. As in some

of the previously commented approaches, such as [FPS01, Car99, Car00], a manually stated test purpose is used to define the desired sequences to be observed on the specification. A synchronous product of the test purpose and the timed automata model is first formed and used to extract a symbolic test sequence with timing constraints that reach a goal state of the test purpose. This symbolic trace can be interpreted at execution time to give a final verdict.

- [RNHW98] gives a particular method for the derivation of the more relevant inputs of systems.

- [PF99] suggests a technique for translating a region graph into a graph where timing constraints are expressed by specific labels using clock zones.

- [RMN08] proposes a logic to infer whether a set of observations (i.e. results of test applications) allows to claim that the `IUT` conforms to the specification if a specific set of hypotheses (taken from a repertory of hypotheses) is assumed. In [RMN08] the soundness and completeness of this new logic with respect to a general notion of conformance is shown. In addition to this new logic, [MNR07a, MNR08a] represent a conservative extension including a complete temporal systems study. The authors adapt some of the [RMN08] rules to cope with this new framework, and they introduce several specific hypotheses and rules to appropriately express time assumptions. In this approach they also prove a correctness result of this new approach with respect to a general notion of timed conformance.

2.3 Summarizing remarks

During the last 15 years several methods have been proposed to tackle the problem of testing timed systems. Each of these proposals faces the problem from a different point of view. The existence of very different approaches is due to the fact that testing timed systems is an inherently difficult task for several reasons, as we commented in the beginning of the chapter. In particular, each studied proposal satisfactorily faces a given issue on the cost of sacrificing a suitable property met by other previous approaches, or leading to new unforeseen challenges. This scenario leaves free room for divergent choices where each one has different benefits and drawbacks. Summarizing, each proposal commented before falls into one or more of the following categories:

(1) Test derivation methods provide test suites that are known to be *uncomplete*. In this case, the value of a constructed test suite is due to one of the following reasons:

   (a) It fully analyzes a *test purpose*, that is a subset of behaviors that are considered specially relevant according to some criteria (e.g., a given functionality provided to the user), leaving the rest of behaviors untested.

   (b) It fulfills a *coverage criterium*, that is, tests in the test suite are designed to fully exercise a given structural characteristic of the specification during testing the `IUT` (e.g., states, locations, edges, etc). Let us note that the corresponding `IUT` structural characteristic is different, in general. So, a coverage criterium is in fact an heuristic *test selection* criterium.

(2) Test derivation methods provide *finite* test suites that are complete. This can be met by adding assumptions about the `IUT` (e.g., determinism or determinizability of systems, output urgency, bounded number of states in the `IUT`, discretizability of time, etc). Usually, the number of required hypotheses is high, or some of them impose very strong limitations.

(3) Test derivation methods provide *infinite* test suites that are complete. In this case, completeness is only met in the limit. Usually, the number of required hypotheses is lower than in the previous alternative, though some hypotheses are still required to reach the property that complete test suites are *countable* sets (e.g., discretizability of time).

Chapter 3

State-of-the-Art: Passive Testing and Monitoring

In this chapter, we briefly expose the more relevant state of the art of passive testing and monitoring systems. An important goal of formal testing is to determine the *conformance*. The activity of conformance testing is essentially focused on verifying the conformity of a given implementation to its specification. In most cases, testing is based on the ability of a tester that stimulates the implementation under test and checks the correction of the answers provided by the implementation [LY96, Lai02]. However, in some situations this activity becomes difficult and even impossible to perform. For example, this is the case if the tester is not provided with a direct interface to interact with the `IUT`. Another conflictive situation appears when the implementation is built from components that are running in their environment and cannot be shutdown or interrupted for a long period of time. In these situations, there is a particular interest in using other types of testing techniques such as *passive testing*.

Passive testing is a testing technique opposite but no incompatible with active testing. The main different between them are that in active testing testers can interact, with any input, with the `IUT` and observe the obtained result in real time. In passive testing approach testers cannot interact directly with the implementation. The usual approach of passive testing consists in recording the trace produced by the implementation under test and trying to find *a fault* by comparing this trace with the specification [LNS+97, Mil98, TC99, TCI99, MA00]. A fault is an abnormal condition or behavior that is different from how the system is either specified or expected to behave.

The problem of fault detection has been studied extensively in the late sixties and early

seventies motivated by testing of sequential circuits and, more recently, by testing of network protocols. A variety of methods have been proposed [Moo56, Hen64, SD88, BU91b, MP93, MP94, YL95, SL95, LY96]. In active testing, generally a test is designed based on the structure of the `FSM` that models the system. Usually, these tests are appropriate for testing an isolated machine of small to medium size. However, for network protocols modeled as `CFSMs`, due to the interactions of component machines and variables, the size of the composite and/or expanded machine is formidable. This makes structured testing impractical. To cope with complexity, unstructured testing has been proposed as well [Wes89, LSKP96]. Almost all the techniques for fault detection in the published literature involve active testing. As we have already mentioned along this Master Thesis, in active testing, the tester has complete control over the inputs and devises a test sequence to reveal possible faults of the system. Obviously, active testing is less applicable to network management. Moreover, testers may have no control over the system inputs and, usually they can only passively observe the input/output behavior. Specifically, in operational networks it is frequently difficult to insert arbitrary inputs without affecting the service or the operation of the network. This naturally leads to passive testing: To observe the input/output behavior of a system in its normal operation for the purpose of detecting faults [Sei72].

Even though passive testing techniques are not new (see for example the approach shown in [AAD79]) in the 1990s a very active research on passive testing was developed. A passive fault detection approach was first proposed in [BHS89]. In this approach passive observers were used for network fault detection. Later, multiple observers were used to reduce the complexity of each observer [WS93]. Other approaches explore relevant properties required for a correct implementation, and then check them on the traces of the systems under test [CGP01, ACN03].

Passive testing has very large domains of application. For instance, it can be used as a *monitoring technique* to detect and report errors. Another area of application is in *network management* to detect configuration problems, fault identification, or resource provisioning (i.e, [MA01, WZY01]). It can be also used to study the feasibility of new features such as classes of services, network security, and congestion control. Nowadays networks are becoming larger, more sophisticated, heterogeneous, and geographically dispersed. In addition, networks are put together by integrating equipment from multiple vendors. Consequently, management of such networks is becoming an important but difficult task. Various things can go wrong, disabling the network or a portion of the network or degrading the performance to an unacceptable level. In many applications, where the end-to-end network performance

needs to be guaranteed, many elements need to be managed at the same time. The complexity of such networks dictates the use of automated network management tools [Tow88]. Fortunately, some standards (i.e, Simple Network Management Protocol v. 3) have been proposed and are becoming popular for communicating status information using a protocol, a database structure specification and a set of data objects. In order to maintain proper operation of a sophisticated network, the system as a whole and each individual component must be in proper working order. In this context, a fault is an abnormal condition or behavior that is different from how the system is either specified or expected to behave. Managing faults in a system is one of the key requirements for network management [Ros81]. The faults that can be managed include:

- Decide if a fault has happened.
- Determine the location of the fault.
- Isolate the rest of the network so that it can continue to function.
- Reconfigure the network to minimize the impact of the fault.
- Repair the fault.

In protocol-system fault detection is often conducted by active testing. A test sequence is designed with a desired fault coverage and then applied to a system implementation under test to reveal faults from its output responses [LY96]. However, for network management, often testers cannot interrupt normal system operations arbitrarily by applying test sequences; testers can only monitor the system input/output behaviors to infer possible faults, and the recommendable testing approach is based in passive testing or passive fault detection [LCH+02]. The global idea is to process the collection traces and apply their correctness with respect to the specification if they owns to the language accepted by specification automaton.

Though passive testing is sometimes mentioned as an alternative to *active testing* [LY96], only little effort has been devoted to this aspect of testing. However, passive testing is worth investigating, since

- Under certain circumstances, it may be the only type of test available, for example in network management.
- It is relatively cheap and easy to implement.
- Active testing is sometimes impractical due to the complexity of systems.

The simplest approach to passive testing makes use of an FSM to model the behavior of the system. In FSM *based passive fault detection*, the specification of the IUT is modeled as an FSM $\mathcal{M}$, and the IUT $\mathcal{N}$ is viewed as a black box where only the execution traces are observable. Accordingly, these execution traces are compared with $\mathcal{M}$ to detect faults in $\mathcal{N}$. The tester wishes to determine whether $\mathcal{N}$ is faulty with respect to $\mathcal{M}$ by observing a sequence t of input/output pairs from $\mathcal{N}$ where the starting state of $\mathcal{N}$ is known or can be estimated. Such a decision can be based on the number of *states that are compatible* with t, being a state s of $\mathcal{M}$ compatible with t if t is a trace of $\mathcal{M}$ starting at s. If the number of states compatible with t is zero then t is sufficient to determine that $\mathcal{N}$ is faulty. Otherwise, t is insufficient to determine whether $\mathcal{N}$ is faulty. It means that there are one or more states compatible with t and t needs to be augmented by an additional input/output sequence of $\mathcal{N}$ to continue with the fault detection.

Normally, authors model communication networks by FSMs [HP92, AP94, BCSS98, NR99, Mea03, CV06]. A transition may be associated with certain input/output behavior, which may not be observable as in the case of internal transitions or networks in which testers can only observe packets but not interactions. Due to limited observability, a network often exhibits certain nondeterministic behavior, which is typically modeled by a nondeterministic finite state machine (NFSM). Additionally, the interactions of different entities in a network can be better modeled by communicating finite state machines (CFSM). Furthermore, since control variables and parameters are usually embedded in the network, authors use extended finite state machines (EFSM) (i.e, [LY96, LNS+97, DU04]). For example, let us suppose that testers have a specification machine $\mathcal{M}$, which models the design or desired behavior of a network. Testers have an implementation machine $\mathcal{N}$, which is the network under test and is a black-box (i.e, testers can only observe its input/output behavior). Passive testing tries to determine whether $\mathcal{N}$ has faults.

There is a pressing need for network management systems capable of handling faults. Sophisticated equipment is much more vulnerable to any single faulty incident. Several approaches are possible for dealing with fault management in modem complex systems and networks. Next we briefly describe some of the most relevant approaches in network monitoring passive testing. One approach is to develop expert systems capable of diagnosing faults and taking corrective action on likely fault scenarios [YWS+89]. The major difficulty here is that the experience of human experts is generally required to develop the expert systems. Each system or subsystem must be handled separately, in general in an ad hoc fashion. In the case of newly developed systems, this may pose problems.

Another approach can be found in [WS93]. This work goal is to look for unifying principles in fault detection and identification. Actually, many network problems that occur due to intrusions and security violations can be addressed by using passive testing. This is clear from the observation that unwanted intrusions matter only if they are successful in changing the input/output behavior of the implemented machine. Thus, many security attacks may be treated as intentional and subtle modifications to the behavior of implemented machine that are manifested only under the presence of certain inputs and when the machine is in a certain state. The goal of [WS93] is to develop a class(es) of fault detection mechanisms that apply broadly across a variety of communication systems. Most communication procedures and systems are currently described in terms of well-defined protocols. These protocols, in turn, are generally specified in terms of discrete-event systems, most commonly `FSM`s. Earlier work along these lines used the concept of a reduced-state `FSM` as an observer, capable of detecting specified types of faults in minimum time. Authors focus on a very simple group of `FSM` observers (generally two states each), capable of detecting almost all possible faults in the system under observation, being exceptions generally deadlock and livelock situations. Thus [WS93] proposes to use a set of independent observers to detect faults in communication systems that are modeled by `FSM`s. An algorithm for constructing these observers and a fast real-time fault detection mechanism used by each observer was given. Since these observers run in parallel and independently, one immediate benefit is that of *graceful degradation*: One failed observer will not cause collapse of the fault management system. In addition, each observer has a simpler structure than the original system and can be operated at higher speed.

In [LS94] the authors proposed an algorithm to trace the variable values as well as the system states, and presented two efficient implementations of the algorithm. In the first implementation, they narrow down the range of each variable as much as possible whenever additional information can be derived from a transition. A set of range operations was introduced and they used examples to illustrate that usage. In the second implementation, the constraints derived from a transition path are recorded and the executability of the path is verified by solving the constraints as a system of linear equations/inequalities. These algorithms can deal with commonly encountered operations on variable values associated with state transitions and also provide efficient variable value determination for the protocol data portion fault detection.

In [LNS+97] also proppose Passive testing for network fault management. In this approach faults are detected in a network protocol system by passively observing its in-

put/output behaviors without interrupting the normal network operations. [LNS+97] introduced methods for passive fault detection of deterministic and nondeterministic FSMs. This work takes into account that it is important for communication networks to detect faults "in-process", that is while the network is in its normal operation. [LNS+97] apply their techniques to management of a signaling network operating under the Signaling System 7 (SS7) and report experimental results, which show the feasibility of applying passive testing to practical systems.

A general formal model for passive conformance testing was first presented in [NVN98], where the FSM is used to model the *protocol control portion*, and fault detection algorithms for both deterministic and nondeterministic systems were designed, implemented, and applied to detect faults at run-time for the Signal System 7 (SS7) protocol system.

In [Mil98, MA00, MA01], the authors specify network systems as CFSMs and study fault detection and location. Backtracking is an effective technique for fault detection [ACC+04]. In [LNS+97] authors developed algorithms for FSM-based passive fault detection. This approach has been applied to other FSM-based systems [WZY01, ZYW01] and was extended to systems specified in the EFSM model by [TC99, LCH+02, CV06, ACC+04, LCH+06] and to systems specified in the CFSM model by [Mil98, MA00, MA01].

[TCI99] presents an extension of the existing algorithms to consider that specification are described as EFSMs. They introduce an algorithm to take into account the number of states and transitions covered. This algorithm is an extension of the one proposed in [LNS+97] to consider EFSMs as the system specifications. Authors experimented, by using in the SDL specification of the GSMMAP. In [TC99] a simple algorithm of passive testing on EFSMs was developed and applied to the GSM-MAP protocol. The algorithm records the values of variables and discards them whenever inconsistency occurs. Yet, no convincing arguments are given on how the faults can be detected.

In [MA01] authors use FSMs model for networks to investigate fault identification using passive testing. The authors illustrate their technique through a simulation of a practical X.25 protocol example.

[ZYW01] studies how passive testing check the protocol implementation through online observation. They showed that passive testing can be performed in the production field without interfering with the network. This test methodology was realized in an OnLine Test System (OLTS). The OLTS exploits the state synchronization algorithm to test the protocol state machine. It tests the exchange and the manipulation of the routing information through the topology analysis and the internal process simulation. OLTS also supports multiple

instances to work in a distributed environment and it can also cooperate with an active test system to bring out the best of each other.

A more systematic study of passive testing of the data portion were reported in [LCH+02]. Variables contain important information of protocol systems. In particular, they determine the system states, and their external behaviors. Let us remark that, it is well known that to test variable values due to the complexity of tracing them [LY96]. In [LCH+02] the authors present two algorithms, using an Event-driven `EFSM`. Experimental results on the Internet routing protocol OSPF were reported. First an effective passive testing algorithm for `EFSM`s was proposed. Second, an algorithm based on variable determination with the constraints on variables was presented. This last algorithm allows us to trace variables values as well as the system state. However, not all transfer errors can be detected. To overcome this limitation, [ACC+04] proposes a new approach based on backward tracing. This algorithm was strongly inspired by [LCH+02], but processes the trace backward in order to further narrow down the possible configurations for the beginning of the trace and to continue the exploration in the past of the trace with the help of the specification. This algorithm contains two phases. First, it follows a given trace backward, from the current configuration to a set of starting ones, according to the specification. The goal is to find the possible starting configurations of the trace, which leads to the current configuration. Then, it analyses the past of this set of starting configurations, also in a backward manner, seeking for end configurations, that is to say configurations in which the variables are determined. When such configurations are reached, they can take a decision on the validity of the studied path. This new algorithm was applied to the Simple Connection Protocol (SCP) that allows to connect two entities after a negotiation of the quality of service required for the connection. The testing results were also compared to the passive testing algorithm in [LCH+02].

In [CV06] the authors propose an approach to passive testing in order to express invariants for network protocols, such as session maintenance protocols. In the proposed technique, critical properties were represented as a set of invariants that an `IUT` should fulfill. Furthermore, the authors propose a mechanism to get around the problem to determine from which state the observation of execution traces started. In order to validate the effectiveness of the proposed approach, the Managed Session Protocol was used as a real-life case study.

In [SL06], authors propose an algorithm, inspired in [LSK+93, LSKP96], where heuristics are used to achieve high coverage of transitions in a `CFSM` model. The authors also studied mutation testing, since it is known to be efficient for a range of particular types of errors in software testing. This approach defines mutation functions with special properties such that

only mutants with single faults need to be considered for test generation. As a case study, authors modeled the predicate (guard) absence fault type FPA with this property, then presented and analyzed the test generation algorithm. The well-known Needham-Schroeder on mutual authentication protocol [NS78, Low96] was used to illustrate their formal model and testing algorithms. In [UXZ07], authors propose a new approach to FSM based passive fault detection which improves the performance of [LNS+97].

Routing protocols are playing an important role for the Internet performance. The protocol testing is an effective means to guarantee the quality of the protocols implementations. Now the most commonly used routing protocols include RIP, OSPF and BGP [HW08] and protocols proposed by other organizations or manufactures. Although the traditional active testing helps a lot to uncover the deficiencies of the implementations of routing protocols [HLSV00, ZYW03], some abnormalities will only appear in practice and/or over long time. The passive testing [LNS+97] can be performed in production filed over a long time without interference on the network. It is able to perform the conformance, interoperability and performance tests. The passive testing only observes online and does not send anything to the IUT. It is quite different from the "injecting input and observing output" way. Thus, corresponding techniques for the passive testing must be worked out. The OnLine Test System (OLTS) was a realization and an application of the passive testing, which made it function in test activities.

Most security protocols use cryptography to achieve data transmission, authentication and key distribution [Mea92, Mea03] in a hostile environment. Several unique characteristics of security protocols make the traditional conformance testing approaches insufficient and pose new challenges for both modeling and test generation tasks. First, security protocols have a huge and special data portion. The input/output messages are from a language defined by cryptographic primitives such as public/private encryption and decryption. The formidable size of the alphabet makes generating a complete checking sequence infeasible. Therefore, tradeoff is usually made to focus only on a special type of non conformance security flaws.

Normally EFSMs can be used to specify the security protocol and augment the model to include security protocol message types as the parameter of input/output symbols. On the other hand, security properties can be tested only with a precise intruder model. In [SL06], authors use EFSM to formally specify the intruder's behaviors based on the well known Dolev Yao model [DY81], which models most powerful and yet realistic intruder. Consequently, the whole protocol system was modeled as the communication system composed of the intruder

and a set of legitimate principals. This approach was extended in [SL06] adding knowledge to the intruders and message confidentiality requirement.

The existence of diverse intruders renders the resilience of those protocol systems more significant, and more challenging. Various formal modeling and analysis techniques, such as *BAN logic*, *model-checking* and *strand spaces* [NS78, Low96, ES00] were developed to ensure the correctness of security protocol system. These works were focused on *validating the protocol specification*. However, errors can also be introduced to the system in implementation phases, even if the specification is proven to be flawless. Furthermore, interconnected communication system interfaces may result in security problems, such as message content exposure.

Systematic testing approaches for security protocols have been largely neglected by the research community, even though numerous reports show programming errors in security-critical systems are very common [Tho03, Tho05]. Testing for system security, often known as *penetration testing* [Tho05] or red-team testing, refers to the activity of executing a predefined test script with the goal of finding a security exploit. Thompson in [Tho03] classified four general penetration testing methods:

- Testing dependency.
- Testing unanticipated user input.
- Expose design vulnerabilities.
- Expose implementation vulnerabilities.

Under these guidelines practical testing has been conducted in industry and proved to be very helpful. Nonetheless, most of the current penetration testing activities are ad-hoc and rely on expert knowledge of target systems or existing exploits [GH02]; the cost of a comprehensive testing is high and the response time is too long. On the other hand, current testing methods are largely at system level on system reconfiguration [SHJ+02] or unexpected side effect of operations [CKXI03]. Protocol level penetration testing has not drawn adequate attention yet is crucial for discovering security protocol implementation errors. Particularly, automated test selection and execution techniques were desirable for complex protocols and for real-time response to security flaws.

In [CGP03] was presented a novel methodology to perform passive testing. The usual approach consists in recording the trace produced by the implementation under test and trying to find a fault by comparing this trace with the specification. This novel approach

was supported by the following idea: A set of invariants represent the most relevant expected properties of the implementation under test. Intuitively, an invariant expresses the fact that each time the implementation under test performs a given sequence of actions, then it must exhibit a behavior reflected in the invariant. Authors propose a more active approach to passive testing where the minimum set of *(critical)* properties required to a correct implementation may be explicitly indicated. In short, an invariant expresses that each time that the implementation under test performs a given sequence of input/output actions, then it must show a behavior respected in the invariant. This approach was able to test the data flow, but not in a very satisfactory way. This was the reason for a second approach seeking to apply a set of constraints to the trace. In order to perform this phase authors present two algorithms based, respectively, on left-to-right and right-to-left pattern matching algorithms. In this approach information was extracted from the specification and then used to process the trace. However, one of the drawbacks of this approach was the limitation on the grammar used to express invariants.

Another approach can be seen in [ACN03]. There, it was proposed that invariants should be supplied by the expert/tester. In this case the first step is to check that the invariant is in fact correct with respect to the specification. An algorithm to check this correctness was provided. The complexity, in the worst case, of the algorithm was linear, with respect to the number of transitions in the specification. Once a set of (correct) invariants was generated the second step consists in checking whether the trace produced by the `IUT` respects the invariants. In order to do so a simple adaptation of the classical algorithms for pattern matching on strings (see e.g. [BM77, KMP77]) was implemented. In [ACN03] was also presented a test architecture for WAP as well as the experimental results obtained from the application of their passive testing with invariants approach. Another experiment with invariant were presented simulating Simple Connection Protocol (SCP) and the results of preliminary experiments are presented in [CGP03].

To improve the fault detection capabilities, [ACC+04] proposes a backward checking method that analyzes in a backward fashion the input/output trace from passive testing and its past. It effectively checks both the control and data portion of a protocol system, compliments the forward checking approaches, and detects more errors. Authors presented their algorithm, studied its termination and complexity, and reported experiment results on the protocol SCP.

In [BCNZ05], authors performed two types of property verification: one on the specification and another one on the implementation. For the first type of verification was developed

algorithms whose complexity were better than classical algorithms for model checking, since these ones were usually exponential on the number of transitions. For the second type of verification authors developed new algorithms that check the properties on the real implementation traces. These algorithms were adaptations of classical algorithms for pattern matching. Let us remark that this kind of verification is not performed at all by model checking. Thus, their techniques are indeed closer to conformance testing or system monitoring than to model checking. In [BCNZ05] was also studied WAP. This protocol is an open global specification that empowers mobile users with wireless devices to easily access and interact with Internet information and services instantly. It is worth to point out that this protocol represents a typical example where active testing cannot be applied since, in general, there is no direct access to the interfaces between the different layers. Thus, testers cannot control how internal communications were established.

In [SL06] was studied testing of message confidentiality and essential security property. The authors formally model protocol systems with an intruder using Dolev-Yao model. The well-known Needham-Schroeder-Lowe protocol is used to illustrate their approaches. In [LCH+06] authors studied network protocol system monitoring for fault detection using a formal technique of passive testing that is a process of detecting system faults by passively observing its input/output behaviors without interrupting its normal operations. After describing a formal model of event-driven `EFSM`s, authors present two algorithms for passive testing of protocol system control and data portions. Experimental results on OSPF and TCP were reported.

In [WSW+07] authors applied a passive testing algorithm to the TCP protocol. Experimental results show that the protocol has a high transition coverage compared with interpretability testing experiments. Detailed analysis of the experiments is present and shows a possible way of combining passive testing and active testing.

In addition to the theoretical framework some authors have developed some software tool. The OnLine Test System (OLTS) [ZYW01]. The prototypes of the OLTS were implemented on both a Sun Ultra 1 Solaris platform and a x86-Linux platform separately to promote the usability. Tcl/TK was exploited to plot a friendly GUI. Besides, they combined both the active testing and the passive testing together. Effective test on routing protocols has been performed.

Another software tool is TestTInv. This tool facilitates the automation of the passive testing approach proposed in [BCNZ05]. In order to test the usefulness of their approach, the tool was exercised in a real-life case study: The Wireless Application Protocol (WAP).

The authors present a test architecture as well as the most relevant results obtained from the application of their approach to the WAP.

Chapter 4

Passive Testing of Timed Systems

The scale and heterogeneity of current systems make impossible for developers to have an overall view of the system. Thus, it is difficult to foresee those errors that are either critical or more probable. Since the construction of a system requires to use several components, developed by different teams, reliability of these components is a must. This is a requirement not only for final customers but also for developers. In this context, *formal testing techniques* provide systematic procedures to check implementations in such a way that the coverage of critical parts/aspects of the system depends less on the intuition of the tester.

The application of formal testing techniques to check the correctness of a system requires to identify its *critical* aspects, that is, those aspects that will make the difference between correct and incorrect behavior. In this line, *the time* consumed by each operation should be considered critical in a real-time system. There has been several proposals for timed testing (e.g. [MMM95, CL97, HNTC99, SVD01, EDK02, ED03, NR06, MNR08c]). In these works, time is considered to be *deterministic*, that is, time requirements follow the form "after/before t time units...". In fact, in most of the cases, time is introduced by means of clocks following [AD94]. Even though the inclusion of time allows the specifier to give a more precise description of the system to be implemented, there are frequent situations that cannot be accurately described by using this notion of deterministic time. For example, we may desire to specify a system where a message is expected to be received with probability $\frac{1}{2}$ in the interval $(0, 1]$, with probability $\frac{1}{4}$ in $(1, 2]$, and so on. In our framework we propose two different ways to take into account time issues represented in specifications. On the one hand, we consider time expressed with fixed values. This approach is close to hardware systems since time is usually considered to have fixed values (i.e, it can be determined by the internal watch, MIPS, and time per cycles of the CPU). On the other hand, in our second

approach we consider that time will be expressed by using probability distribution functions. This approach is closer to software systems where the time to execute some input/output is close to a fix number, with probability p, but it would not be credible to give a fixed time value since this time may depend on several uncontrollable effects.

Along this chapter we present two formal passive testing frameworks where the temporal behaviour of systems is considered. Two extensions of the classical concept of *Finite State Machine* will allow a specifier to explicitly denote temporal requirements for each action of a system. Intuitively, transitions in finite state machines indicate that if the machine is in a state s and receives and input i then it will produce and output o and it will change its state to s'. An appropriate notation for such a transition could be $s \xrightarrow{i/o} s'$. In contrast if we consider an extension of finite state machines to consider fix time, transitions such as $s \xrightarrow{i/o}_t s'$ indicate that the time between receiving the input i and returning the output o is given by t.

However, these are some similarities. We will separately present the two testing methodologies since the treatment of time in both settings is very different, an specific approaches have to be used in each case. In both cases, we consider a set of observations collected by means of the interaction with the implementation (i.e, logs of the system) and establish different temporal properties associated with the invariants.

As we have already explained in the first chapter of this Master Thesis, in this work we represent time properties with invariants. In addition we provide algorithms to decide the correctness of the invariants with respect to the specifications and algorithms to decide if an invariant detect faults with respect to the traces generated by the implementation. Let us remark that, due to the fact that we consider a black-box testing framework, testers cannot compare in a direct way timed requirements of the *real* implementation with those established in the specification.

The rest of the chapter is organized as follows. In Section 4.1 we introduce additional notation used along the chapter such as the notion of time interval, $\texttt{TFSM}_{st}$,and $\texttt{TFSM}_{ft}$. In Section 4.2 we present our first novel approach framework in passive testing where time is expressed by fix values. In Section 4.3 we present our second novel approach framework, where time is given by probability distribution functions.

4.1 Preliminaries

Along this work, we consider that time values belong to a generic domain $\mathcal{T}$. Most concepts will be parameterized with respect to this domain. However, some of the notions

and definitions will depend on the specific instance of the generic time domain. Specifically, we will consider three different possibilities to represent time: Time values, stochastic time, and time intervals.

Event though specifications will use either fix time values of probability distribution functions, we will use time intervals within the definition of some invariants.

Definition 4.1 We say that any value $t \in \mathbb{R}_+$ is a *fixed time value*. We say that $\hat{a} = [a_1, a_2]$ is a *time interval* if $a_1 \in \mathbb{R}_+$, $a_2 \in \mathbb{R}_+ \cup \{\infty\}$, and $a_1 \leq a_2$. We assume that for all $t \in \mathbb{R}_+$ we have $t < \infty$ and $t + \infty = \infty$. We consider that $\mathcal{IR}$ denotes the set of time intervals. Let $\hat{a} = [a_1, a_2]$ and $\hat{b} = [b_1, b_2]$ be time intervals. We consider the following functions:

- $\oplus : \mathcal{IR} \times \mathbb{R}_+ \rightarrow \mathcal{IR}$ defined as $\oplus(\hat{a}, t) = [a_1 + t, a_2 + t]$.
- $\boxminus : \mathcal{IR} \times \mathcal{IR} \rightarrow \mathcal{IR}$ defined as $\boxminus(\hat{a}, \hat{b}) = [\min(a_1, b_1), \max(a_2, b_2)]$, where min and max denote the minimum and maximum value respectively.
- $+ : \mathcal{IR} \times \mathcal{IR} \rightarrow \mathcal{IR}$ defined as $[a_1, a_2] + [b_1, b_2] = [a_1 + b_1, a_2 + b_2]$.
- $\subseteq : \mathcal{IR} \times \mathcal{IR} \rightarrow \{true, false\}$ defined as $[a_1, a_2] \subseteq [b_1, b_2] = (a_1 \geq b_1 \wedge a_2 \leq b_2)$.
- $\odot : \mathcal{IR} \times \mathbb{R}_+ \rightarrow \{true, false\}$ defined as $[a_1, a_2] \odot t = (t \leq a_2)$.

□

Fixed time values are used to express precise moments where a signal is processed by a system. For example, if we associate t time units with a transition, then always this transition is performed in t time units. This representation is very useful, for example in security protocols, where some signals must be sent exactly every 2 seconds. Time intervals will be used to express time constraints associated with the execution of actions. The idea is that if we associate a time interval $[t_1, t_2] \in \mathcal{IR}$ with a task we indicate that this task should take at least t_1 time units and at most t_2 time units to be performed. Intervals like $[0, t]$, $[t, \infty)$, or $[0, \infty)$ denote the absence of a temporal lower/upper bound and the absence of any bound, respectively.

Next we introduce the concepts of probability distribution function and confidence.

Definition 4.2 A *probability distribution function* is a function $F : \mathbb{R}_+ \longrightarrow [0, 1]$ having the following properties:

- $\lim_{t \rightarrow +\infty} F(t) = 1$.

- F is monotonically increasing, that is, for all t_1 and $t_2 \in \mathbb{R}_+$ such that $t_1 \leq t_2$ we have $F(t_1) \leq F(t_2)$.

- F is a right-continuous function at any point, that is, for all $t \in \mathbb{R}_+$ we have:

$$\lim_{t' \to t^+} F(t') = F(t).$$

We denote the set of probability distribution functions by $\mathcal{F}$ (F, F_1, F_2 to range over $\mathcal{F}$). Let F_1 and F_2 be two probability distribution functions. We write $F_1 = F_2$ if for all $t \in \mathbb{R}_+$ we have $F_1(t) = F_2(t)$. We will call *sample* to any multiset of positive real numbers. We denote the set of multisets in $\mathbb{R}_+$ by $\wp(\mathbb{R}_+)$. Let F be a probability distribution function and J be a sample. We denote the *confidence* of F on J by $\gamma(F, J)$. □

In our setting, samples will be associated with time values that implementations need to perform sequences of actions. We have that $\gamma(F, J)$ takes values in the interval $[0, 1]$. Intuitively, bigger values of $\gamma(F, J)$ indicate that the observed sample J is more likely to be produced by the probability distributed function F. That is, γ decides how *similar* the probability distribution function generated by J and the one corresponding to F are.

Next, we introduce one of the standard ways to measure the confidence degree that a probability distribution function F has on a sample. In order to do so, we will present a methodology to perform *hypothesis contrasts*. The underlying idea is that a sample will be *rejected* if the probability of observing that sample from a *natural* sample extracted from F is low. In practice, we will check whether the probability to observe a *discrepancy* lower than or equal to the one we have observed is low enough. We will present *Pearson's* χ^2 *contrast*.

Definition 4.3 The *Pearson's* χ^2 *contrast* can be applied both to continuous and discrete probability distribution functions. Once we have collected a sample of size n we perform the following steps:

- We split the sample into k classes which cover all the possible range of values. We denote by o_i the *observed frequency* at class i (i.e. the number of elements belonging to the class i).

- We calculate the probability p_i of each class, according to the proposed probability distribution function. We denote by e_i the *expected frequency*, which is given by the equation $e_i = n \cdot p_i$.

- We calculate the *discrepancy* between observed frequencies and expected frequencies as $X^2 = \sum_{i=1}^{k} \frac{(o_i - e_i)^2}{e_i}$. When the model is correct, this discrepancy is approximately distributed as the distribution χ^2.

- We estimate the number of freedom degrees of χ^2 as $k - r - 1$. In this case, r is the number of parameters of the model which have been estimated by maximal likelihood over the sample to estimate the values of p_i (i.e. $r = 0$ if the model completely specifies the values of p_i before the samples are observed).

- We will *accept* that the sample follows the proposed random variable if the probability to obtain a discrepancy greater or equal to the discrepancy observed is high enough, that is, if $X^2 < \chi^2_\alpha(k - r - 1)$ for some α low enough. Actually, as such margin to accept the sample decreases as α decreases, we can obtain a measure of the validity of the sample as $\texttt{max}\{\alpha \mid X^2 < \chi^2_\alpha(k - r - 1)\}$.

□

Example 4.1 Let us illustrate the previous definitions of probability distribution function, sample, confidence and Person χ^2 contrast in the following example. Let us suppose we have a dice, having its six sides the same probability. We represent this probability function in Figure 4.1. For example, the probability of obtaining 1 or less is $\frac{1}{6}$, and the probability of obtaining a number less than or equal to 4 is $\frac{4}{6}$.

Suppose we toss this dice three hundred times. We store the observed results in a sample denoted by ℓ. We have that ℓ is in $\wp(\{1, 2, 3, 4, 5, 6\})$. In order to represent the number of observed values associated which each side of the dice, let us suppose that in ℓ the observed frequencies are $o_1 = 43$, $o_2 = 49$, $o_3 = 56$, $o_4 = 45$, $o_5 = 66$, and $o_6 = 41$.

Now we show how can we decide the confidence of ℓ with respect to the function represented in Figure 4.1. We will use for this task the *chi square goodness of fit test*. This test is particularly useful to determine how well a model fits observed data since it allows us to evaluate how *close* the observed values are to those which would be expected given the model in question.

We denote the expected frequency of value i by e_i. Since we expected that the dice is regular, we have $e_i = 50$ with $1 \leq i \leq 6$.

The level of significance $\alpha \in [0, 1]$ allows us to let some discrepancies in the values with respect to the expected ones. We define the null hypothesis, denoted by H_0, and we must show that H_0 does not hold. The meaning of the null hypothesis in this example is "the dice is not regular", meaning that $F(x) \neq \frac{x}{6}$, for some $x \in \{1, \ldots, 6\}$.

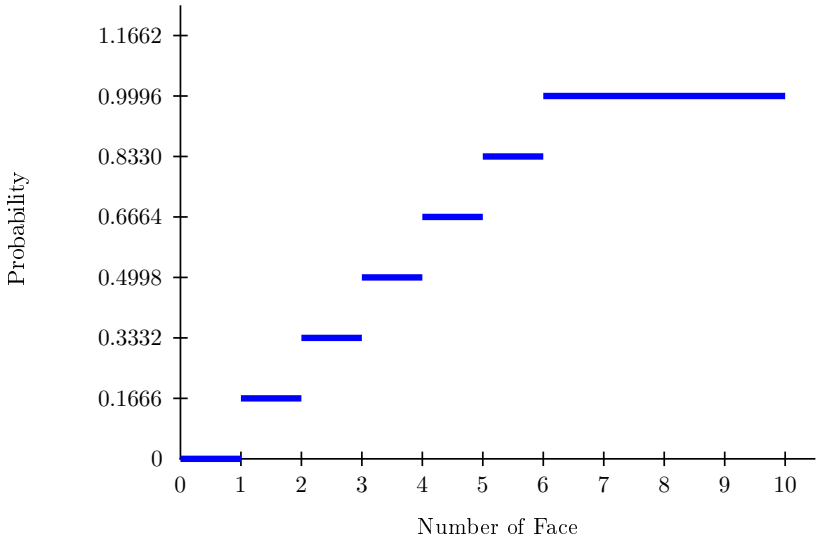


Figure 4.1: Probability distribution function of a regular dice.

$$H_0 = \sum_{i=1}^{6} \frac{(o_i - e_i)^2}{e_i} > \chi^2_{5;\alpha}$$

If we use $\alpha = 0.05$ the we are saying "H_0 does not hold the with probability $1 - \alpha$", in other words that with probability $1 - \alpha$, ℓ was obtained from F. In our case, we can accept that the dice is regular because H_0 does not hold. □

Next we introduce our two timed extensions of the classical finite state machine model. The main differences with respect to usual FSMs consists in the addition of *time* to indicate the lapse between offering an input and receiving an output.

Definition 4.4 A *Fixed Timed Finite State Machine*, in the following $\texttt{TFSM}_{ft}$, is a tuple $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$ where S is a finite set of states, $\mathcal{I}$ is the set of input actions, $\mathcal{O}$ is the set of output actions, Tr is the set of transitions, and s_{in} is the initial state.

A transition belonging to Tr is a tuple (s, s', i, o, t) where $s, s' \in S$ are the initial and final states of the transition, $i \in \mathcal{I}$ and $o \in \mathcal{O}$ are the input and output actions, respectively, and $t \in \mathbb{R}_+$ denotes the time that the transition needs to be completed. We say that M is *input-enabled* if for all state $s \in S$ and input $i \in I$, there exist $s' \in S$, $o \in O$, and $t \in \mathbb{R}_+$ such that $(s, s', i, o, t) \in Tr$. □

Intuitively, a transition (s, s', i, o, t) of a $\texttt{TFSM}_{ft}$ indicates that if the machine is in state s and receives the input i then, after t time units, the machine emits the output o and moves to s'. We denote this transition by $s \xrightarrow{i/o}_t s'$.

Definition 4.5 A *Stochastic Timed Finite State Machine*, in the following $\texttt{TFSM}_{st}$, is a tuple $M = (S, I, O, Tr, s_{in})$ where S is a finite set of states, I is the set of input actions, O is the set of output actions, Tr is the set of transitions, and s_{in} is the initial state.

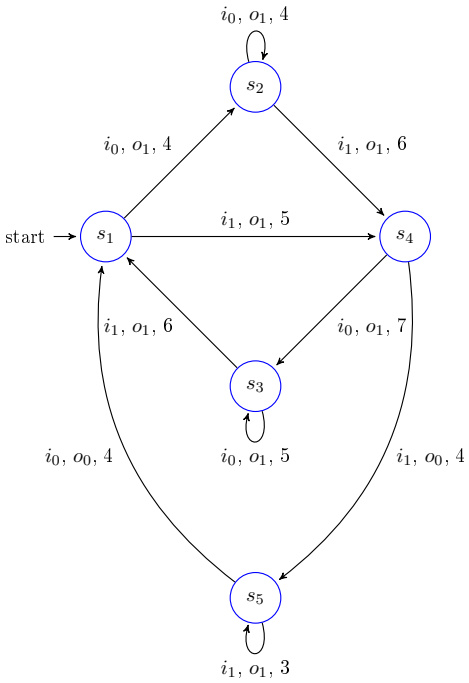
A transition belonging to Tr is a tuple (s, s', i, o, F) where $s, s' \in S$ are the initial and final states of the transition, $i \in I$ and $o \in O$ are the input and output actions, respectively, and $F \in \mathcal{F}$ denotes the time, in probability terms, that the transition needs to be completed.

We say that M is *input-enabled* if for all state $s \in S$ and input $i \in I$, there exist $s' \in S$, $o \in O$, and $F \in \mathcal{F}$ such that $(s, s', i, o, F) \in Tr$. We say that M has *regular stochastic information*, if there do not exist two different transitions (s, s', i, o, F_1) and (s_1, s_2, i, o, F_2) with $F_1 \neq F_2$. □

Intuitively, a transition (s, s', i, o, F) of a $\texttt{TFSM}_{ft}$ indicates that if the machine is in state s and receives the input i then, after a lapse of t time units generated from the probability distribution function F, the machine emits the output o and moves to s'. We usually denote such transition by $s \xrightarrow{i/o}_F s'$. Along the rest of the work we assume that all the machines are observable non-deterministic and in the case of $\texttt{TFSM}_{st}$ we assume that all machines have regular stochastic information.

Example 4.2 In this example we briefly describe the behaviour of the $\texttt{TFSM}_{ft}$ represented in Figure 4.2 and of the $\texttt{TFSM}_{st}$ represented in Figure 4.3. In Figure 4.2 we give a graphical representation of a TFSM where s_1 is the initial state. We can observe different transition such as $s_1 \xrightarrow{i_0/o_1}_4 s_2$. Let us note that, according to the definition, all time values are in $\mathbb{R}_+$ but, in contrast with the $\texttt{TFSM}_{st}$ model time values are not uniquely associated with an input/output pair. For example, we have two transitions $s_1 \xrightarrow{i_1/o_1}_5 s_4$ and $s_2 \xrightarrow{i_1/o_1}_6 s_4$ having the same associated input/output pair but different time values.

Let us consider the $\texttt{TFSM}_{st}$ depicted in Figure 4.3. We are modelling the timed behavior, with the probability distribution functions F_1, F_2, F_3 associated with each transition (In Figure 4.4 we show a graphical representation of these three functions). In this example we show three possible, often used, probability distribution functions. For instance, we may consider that the all values generated by the F_1 function are *uniformly distributed* in the interval $[0, 2]$. Uniform distributions allow us to keep compatibility with time intervals in (non-stochastic) timed models in the sense that the same *weight* is assigned to all the times



$$I = \{i_0, i_1\}$$

$$O = \{o_0, o_1\}$$

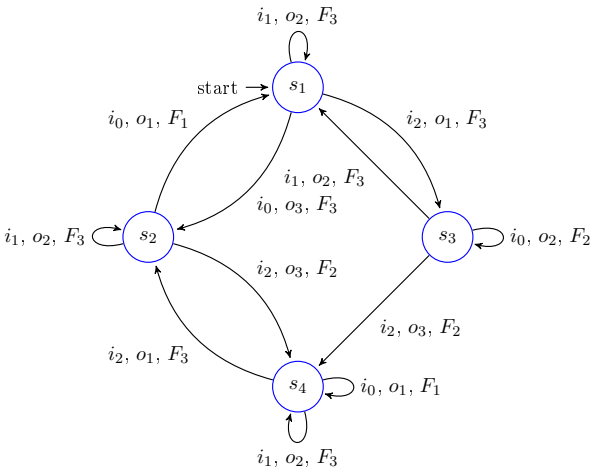
Figure 4.2: Example of $\texttt{TFSM}_{ft}$.

in the interval. We consider that F_2 follows a Dirac distribution in 4. The Dirac distribution concentrates all the probability in a single point. Thus a Dirac distribution in n gives probability 1 to n and probability 0 to the rest of values. In timed terms, the idea is that the corresponding delay will be equal to n time units. Dirac distributions allow us to simulate deterministic delays appearing in timed models. Finally, F_3 is *exponentially* distributed with parameter 3.

For instance, let us consider the transition t_{23}. Intuitively, if the machine is in state 2 and receives the input i_0 then it will produce the output o_1 after a time given by F_1 and will move to state 3. The time associated with the transition is a value $0 \leq t \leq 2$, that can be drawn with the same probability. □

During the rest of this Master Thesis, in definitions where the introduced concepts are the same for $\texttt{TFSM}_{ft}$ or $\texttt{TFSM}_{st}$ we will use the generic name TFSM. Next, we introduce the notion of *trace* of a TFSM. As usual, a trace is a sequence of input/output pairs. In addition, we have to record the time that the trace needs to be performed.

Traces are essential in passive testing. Let us remember that, in our setting, testers cannot interact with the IUT. They are only provided with recorder traces, called *logs*, for making testing. In a log we can observe several signals (inputs/output) and the time where



$$F_1(t) = \begin{cases} 0 & \text{if } t \leq 0 \\ \frac{t}{2} & \text{if } 0 < t < 2 \\ 1 & \text{if } t \geq 2 \end{cases}$$

$$F_2(t) = \begin{cases} 0 & \text{if } t < 4 \\ 1 & \text{if } t \geq 4 \end{cases}$$

$$F_3(t) = \begin{cases} 1 - e^{\frac{-t}{3}} & \text{if } t \geq 0 \\ 0 & \text{if } t < 0 \end{cases}$$

$$I = \{i_0, i_1, i_2\}$$

$$O = \{o_1, o_2, o_3\}$$

Figure 4.3: Example of $\texttt{TFSM}_{st}$.

they were performance. A log is a finite sequence and it will look like

$$i_1/o_1/t_1, i_2/o_2/t_2, i_3/o_3/t_3, \ldots, i_n/o_n/t_n$$

Depending on the notion of time represented in the considered machines, testers will consider one approach or the other to decide the validity of the trace recorded form the IUT.

4.2 Fixed Time Approach

In this section we introduce the notion of timed invariant for machines representing time as fix values. For example, we can express that the time the system takes to perform a transition always belongs to a specific interval. Thus, timed invariants are used to express the temporal restrictions of a trace. In our formalism, we assume that timed invariants are given by the tester, possibly derived from the original requirements. Another approach is to consider that they are extracted from the specification. In fact, we can do this easily by adapting the method given in [CGP03] to our timed framework. However, this leads to a huge set of invariants, being most of them irrelevant. In our approach we need to check that the timed invariants proposed by the tester are correct with respect to the specification. Once we have a collection of correct timed invariants, we will have to check if these invariants

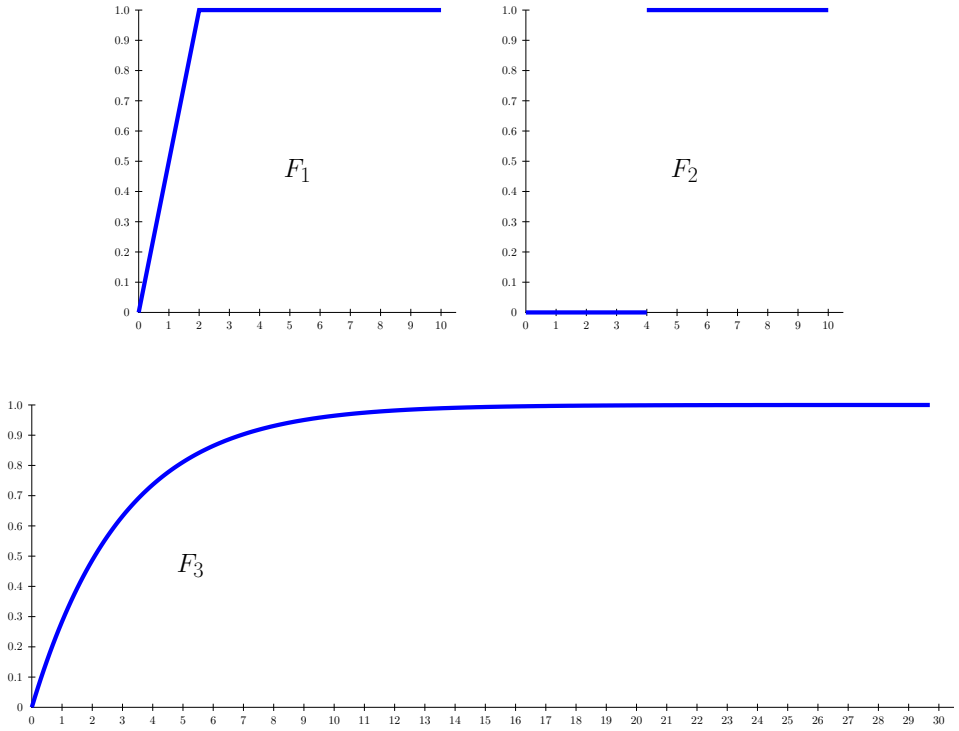


Figure 4.4: Representation of probability distribution functions F_1, F_2, F_3.

are satisfied by the traces produced by the implementation. We will provide an algorithm to verify the correctness of the log, recorded from the implementation, with respect to an invariant.

In order to express traces in a concise way, we will use the wild-card characters ? and $\star$. The wild-car ? represents any value in the sets I and O, while $\star$ represents a sequence of input/output pairs.

Definition 4.6 Let $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$ be a $\texttt{TFSM}_{ft}$. We say that the sequence I is a *fix time invariant* for M if the following two conditions hold:

1. I is defined according to the following EBNF:

$$I ::= a/z/\hat{p}, I \mid \star/\hat{p}, I' \mid i \mapsto O/\hat{p} \rhd \hat{t}$$
$$I' ::= i/z/\hat{p}, I \mid i \mapsto O/\hat{p} \rhd \hat{t}$$

   In this expression we consider $\hat{p}, \hat{t} \in \mathcal{IR}$, $i \in \mathcal{I}$, $a \in \mathcal{I} \cup \{?\}$, $z \in \mathcal{O} \cup \{?\}$, and $O \subseteq \mathcal{O}$.

2. I is *correct* with respect to M.

We denote the set of fixed timed invariants by FixedTimeInv. □

Let us remark that time conditions established in invariants are given by intervals. However, machines in our formalism present time information expressed as fix amounts of time. This fact is due to consider that it can be admissible that the execution of a task sometimes lasts more than expected: If most of the times the task is performed on time, a small number of delays can be tolerated. Moreover, another reason for the tester to allow imprecisions is that the artifacts measuring time while testing a system might not be as precise as desirable. In this case, an apparent wrong behavior due to bad timing can be in fact correct since it may happen that the *watches* are not working properly. A longer explanation on the use of time intervals to deal with imprecisions can be found in [MNR07b].

Intuitively, the previous EBNF expresses that an invariant is either a sequence of symbols where each component, but the last one, is either an expression $a/z/\hat{p}$, with a being an input action or the wild-card character ?, z being an output action or the wild-card character ?, and $\hat{p}$ being a timed interval, or an expression $\star/\hat{p}$. There are two restrictions to this rule. First, an invariant cannot contain two consecutive expressions $\star/\hat{p}_1$ and $\star/\hat{p}_2$. In the case that such situation was needed to represent a property, the tester could simulate it by means of the expression $*, (\hat{p}_1 + \hat{p}_2)$. The second restriction is that an invariant cannot present a component of the form $\star/\hat{p}$ followed by an expression beginning with the wildcard character ?, that is, the input of the next component must be a *real* input action $i \in \mathcal{I}$. In fact, $\star$ represents any sequence of input/output pairs such that the input is not equal to i, being i the next input appearing in the invariant.

The last component, corresponding to the expression $i \mapsto O/\hat{p} \rhd \hat{t}$, is an input action followed by a set of output actions and two timed restrictions, denoted by means of two intervals $\hat{p}$ and $\hat{t}$. The first one is associated to the last expression of the sequence. The second one is related to the sum of time values associated to all input/output pairs performed before. For example, the meaning of an invariant as $i/o/\hat{p}, \star/\hat{p_\star}, i' \mapsto O/\hat{p'} \rhd \hat{t}$ is that if we observe the transition i/o in a time belonging to the interval $\hat{p}$, then the first occurrence of the input symbol i' after a lapse of time belonging to the interval $\hat{p_\star}$, must be followed by an output belonging to the set O, in a time belonging to $\hat{p'}$. The interval $\hat{t}$ makes reference to the total time that the system must spend to perform the whole trace. This notion of invariant allows us to express several properties of the system under study. Next, we introduce some examples in order to present how invariants work.

Example 4.3 The simplest invariant we can define within our framework follows the scheme $i \mapsto \{o\}/[2,3] \rhd [2,3]$. The idea is that each occurrence of the symbol i is followed by the output symbol o and this transition is performed between 2 and 3 time units.

We can specify a more complex property by taking into account that we are interested in observing the output o after the input i only if the input i_0 was previously observed. In addition, we include intervals corresponding to the amount of time the system takes for each of the transitions and the total time it spends in the whole trace. We could express this property by means of the invariant $i_0/?/[1,4],\star/[0,5], i \mapsto \{o\}/[2,3] \rhd [2,12]$. An observed trace will be correct with respect to this invariant if each time that we find a (sub)sequence starting with the input i_0 and any output symbol which has been performed in an amount of time belonging to the interval $[1,4]$, then if there is an occurrence of the input symbol i before 5 time units pass then the input i must be paired with the output symbol o and the lapse between i and o must be in the interval $[2,3]$. In addition, the whole sequence must take a time belonging to the interval $[2,12]$.

We can refine the previous invariant if we consider only the cases where the pair i_0/o_0 was observed. The invariant for denoting this property is the following $i_0/o_0/[1,4],\star/[0,5], i \mapsto \{o\}/[2,3] \rhd [2,12]$. Let us remark that we could not deduce that we have found an error if the pair i_0/o_0 appears in the observed trace but the input i is not detected afterwards in the corresponding trace. In such a situation we cannot conclude that the implementation fails. Similarly, if we find the pair i_0/o_1 we cannot conclude anything since the premise of the invariant, that is, the whole sequence but the last pair was not found. An invariant as $i \mapsto \{o_1, o_2\}/[1,4] \rhd [1,4]$ indicates that after input i we observe either o_1 or o_2 in a time belonging to [1,4]. □

Since we assume that invariants can be defined by a tester, we must ensure that they are correct with respect to the specification. Next we explain the most relevant aspects of our algorithm to decide whether an invariant is correct with respect to a specification. We separate the algorithm into three different parts. The first part of the algorithm (see Figure 4.5) is responsible for treating the *preface* of the invariant, that is, to determine the states that can be reached in the specification after the first $n-1$ input/output/time tuples have been traversed. The second phase (see Figure 4.6) is used to check that the last pair of the invariant is correct for the specification. In other words, to detect that for all the states computed in the previous step, if the last input of the invariant can be performed then the obtained output belongs to the set of outputs appearing in this last expression of the invariant. In addition we also check that these transitions are performed in the time interval appearing in the invariant. Finally, the third part of the algorithm (see Figure 4.7) verifies the last part of the invariant: The sequence is always performed in a time belonging to the corresponding interval. Next we introduce additional notation.

Definition 4.7 Let $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$ be a $\mathtt{TFSM}_{ft}$, $s \in S$, $a \in \mathcal{I} \cup \{?\}$, $z \in \mathcal{O} \cup \{?\}$, and $\hat{t} \in \mathcal{IR}$. We define the set $\mathtt{afterCond}(s, a, z, \hat{t})$ as the set of transitions belonging to Tr having as initial state s, as input a, as output z, and such that its time belongs to the interval $\hat{t}$.

$$\mathtt{afterCond}(s, i, o, \hat{t}) = \{(s, s', i, o, t) | \exists s' \in S, t \in \mathbb{R}_+ (s, s', i, o, t) \in Tr \wedge t \in \hat{t}\}$$

$$\mathtt{afterCond}(s, ?, o, \hat{t}) = \bigcup_{i \in \mathcal{I}} \mathtt{afterCond}(s, i, o, \hat{t})$$

$$\mathtt{afterCond}(s, i, ?, \hat{t}) = \bigcup_{o \in \mathcal{O}} \mathtt{afterCond}(s, i, o, \hat{t})$$

$$\mathtt{afterCond}(s, ?, ?, \hat{t}) = \bigcup_{i \in \mathcal{I}, o \in \mathcal{O}} \mathtt{afterCond}(s, i, o, \hat{t})$$

We define the function $\mathtt{afterInt}(s, \hat{t}, i)$ as the function that computes the set of pairs (s', t) of states $s' \in S$ that can be reached from state s after t time units, belonging t to the interval $\hat{t}$, and such that the input i is not performed.

We will use an auxiliary function so that $\mathtt{afterIntAux}(s, \hat{t}, i) = \mathtt{afterIntAux}(s, \hat{t}, i, 0)$, being this function defined as follows:

$$\mathtt{afterIntAux}(s, \hat{t}, i, tot) = \{(s, tot) | tot \in \hat{t}\} \cup \bigcup_{\substack{(s, s'', i', o, t) \in Tr \\ \hat{t} \odot (tot + t) \\ i \neq i'}} \mathtt{afterIntAux}(s'', \hat{t}, i, tot + t)$$

□

In the first phase of the algorithm we have to initially obtain the set of states that can perform the first input/output pair of the invariant. We compute the states that can be reached from that initial set after performing that transition and such that the time value associated with the transition falls within the range marked by the invariant. We iterate this process until we reach the last expression of the invariant. It is worth to point out that instead of implementing the traversal of the invariant by incrementing a counter, we consider two auxiliary functions: `head()` returns the first element of the invariant and `tail()` removes it. Let us remark that we distinguish between input/output pairs, possibly including the

in : $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$.
$I = \{a_1/\hat{p_1}, \ldots, a_{n-1}/\hat{p_{n-1}}, i_n \mapsto O/\hat{p_n} \rhd \hat{p}\}$
// *where for all* $1 \leq k \leq n-1$ *we have that* $\hat{p}_k \in \mathcal{IR}$,
// *and either* $a_k = i_k/o_k$, *with* $i_k \in \mathcal{I} \cup \{?\}$ *and* $o_k \in \mathcal{O} \cup \{?\}$, *or* $a_k = \star$;
// $i_n \in \mathcal{I}$, $O \subseteq \mathcal{O}$, *and* $\hat{p_n}, \hat{p} \in \mathcal{IR}$.
out: `Bool`.
b :: array of $\mathcal{IR}[|S|]$;
// *an array containing time intervals, having size* $|S|$,
// *and being* $\perp$ *the initial value of all positions*
$I' = I$; $S' \leftarrow S$; $j \leftarrow 1$; $S'' \leftarrow \emptyset$;
while $(j < n)$ **do**
  b' :: array of $\mathcal{IR}[|S|]$;
  if $(\texttt{head}(I') = (\star/\hat{t}))$ **then**
    while $(S' \neq \emptyset)$ **do**
      `Choose` $s_\alpha \in S'$; $S' \leftarrow S' \setminus \{s_\alpha\}$; $ST \leftarrow \texttt{afterInt}(s_\alpha, \hat{t}, i_{j+1})$;
      while $(ST \neq \emptyset)$ **do**
        `Choose` $(s_p, t) \in ST$; $ST \leftarrow ST \setminus \{(s_p, t)\}$; $S'' \leftarrow S'' \cup \{s_p\}$;
        if $(b'_p = \perp)$ **then**
          $b'_p \leftarrow \oplus(b_\alpha, t)$;
        else
          $b'_p \leftarrow \boxminus(\oplus(b_\alpha, t), {b_p}')$;
  else
    while $(S' \neq \emptyset)$ **do**
      `Choose` $s_a \in S'$; $S' \leftarrow S' \setminus \{s_a\}$; $Tr' \leftarrow \texttt{afterCond}(s_a, i_j, o_j, \hat{p_j})$;
      while $(Tr' \neq \emptyset)$ **do**
        `Choose` $(s_a, s_b, i_j, o_j, t) \in Tr'$; $Tr' \leftarrow Tr' - \{(s_a, s_b, i_j, o_j, t)\}$;
        if $(b'_b = \perp)$ **then**
          $b'_b \leftarrow \oplus(b_a, t)$;
        else
          $b'_b \leftarrow \boxminus(\oplus(b_a, t), {b_b}')$;
        $S'' \leftarrow S'' \cup \{s_b\}$;
  $I' = \texttt{tail}(I')$; $b \leftarrow b'$; $S' \leftarrow S''$; $S'' \leftarrow \emptyset$; $j \leftarrow j + 1$;

Figure 4.5: Correctness of an invariant in FixedTimeInv with respect to a specification (1/3).

```
error ← false;
if (S' = ∅) then
    error ← true;
end
b' :: array of IR[|S|];
while (S' ≠ ∅) do
    Choose s_a ∈ S';
    S' ← S' \ {s_a};
    Tr' ← afterCond(s_a, i_n, ?, [0, ∞]);
    while (Tr' ≠ ∅) do
        Choose (s_a, s_b, i_n, o, t) ∈ Tr';
        Tr' ← Tr' \ {(s_a, s_b, i_n, o, t)};
        if ((o ∈ O) ∧ (t ∈ p̂_n)) then
            if (b'_b = ⊥) then
                b'_b ← ⊕(b_a, t);
            else
                b'_b ← ⊟(⊕(b_a, t), b'_b);
            S'' ← S'' ∪ {s_b};
        else
            error ← true
```

Figure 4.6: Correctness of an invariant in FixedTimeInv with respect to a specification (2/3).

wild-character ?, and occurrences of $\star$. In the latter case we will use the previously defined `afterInt`() function to compute the corresponding reached states.

The *input* of the second phase of the algorithm (see Figure 4.6) is the set of states that can be reached after the preface of the invariant is performed. In addition, we also record the time that it took to reach each of these states. If this set is empty then the invariant is not correct. The idea is that we should not use an invariant such that its sequence of input/output/interval cannot be performed in the specification. If this set is not empty, we will check that for all reached states if they can perform the last input of the invariant then the obtained output must belong to the set of outputs appearing in this last expression of the invariant. In addition, time values have to belong to the time interval of the invariant.

The third step of the algorithm (Figure 4.7) will be devoted to check that the time behavior of the whole invariant is correct with respect to the specification. In order to do

```
if (S'' = ∅) then
    error ← true;
end
while (S'' ≠ ∅) do
    Choose s_i ∈ S'';
    S'' ← S'' \ {s_i};
    if (¬(b'_i ⊆ p̂)) then
        error ← true;
return (¬error);
```

Figure 4.7: Correctness of an invariant in FixedTimeInv with respect to a specification (3/3).

this, in the previous stages we recorded all the time values associated with the performance of input/output pairs. We use the functions $\oplus$ and $\boxminus$ to operate with the recorded time values and construct an interval. Thus, in the position k of the array b we store an interval that has as bounds the minimal/maximal times that are needed to reach the state k after performing the whole invariant. If a state is not reachable after the sequence associated with the invariant then $b[k] = \bot$. Next, we concentrate only in states of the specification that can be reached, that is, $b[k] \neq \bot$ and check that all those intervals are contained in the interval appearing at the very last position of the invariant.

Lemma 4.1 Let $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$ be a $\texttt{TFSM}_{ft}$. The worst case of the algorithm given in Figures 4.5, 4.6 and 4.7 checks the correctness of a given invariant in FixedTimeInv $I = i_1/o_1/\hat{p}_1, \ldots, i_{n-1}/o_{n-1}/\hat{p}_{n-1}, i_n \mapsto O/\hat{p}_n \rhd \hat{t}$ with respect to M:

- In time $\mathcal{O}(n \cdot |Tr| \cdot |S|)$ and space $\mathcal{O}(|Tr| + |S|)$ if I does not present occurrences of $\star$.
- In time $\mathcal{O}(k \cdot |Tr|^2 + (n-k) \cdot |Tr| \cdot |S|)$ and space $\mathcal{O}(|Tr| + |S|)$ if I presents occurrences of $\star$, being k the number of $\star$'s in I.

□

Conformance of traces with respect to invariants

In this section we proceed to determine whether the trace obtained from the implementation satisfies the properties indicated by the timed invariants that we are interested in. Let us comment a very important difference with respect to previous proposals for passive

testing: A *homing* state phase (that is, to identify when the sequence was passing by the initial state) is not needed for this kind of invariants. This is so because invariants have to be fulfilled at any point of the implementation. Thus, it is not relevant the state where the machine was placed when we started to observe the trace. In order to test the trace we perform a pattern matching strategy. We have implemented an adaption of the classical algorithms for pattern matching on strings, (i,e. [BM77, KMP77]). We have to consider, for an invariant of length n, all the occurrences of the first $n-1$ elements in the trace. In addition to match the pairs of input/output actions presented in the invariant, the times that are recorded in the trace must belong to the corresponding time intervals in the invariant. Then, if we find a pair i/o such that $i_n = i$ then we have to check that $o \in O$. Finally, we must check the timed restrictions for the last pair of actions and the whole trace. We can say that we have found a mismatch (that is, a fault) if this last condition does not hold.

Next, we explain the main features of the algorithm that we use to establish the conformance of a trace obtained from the `IUT` with respect to an invariant. We present the core of the algorithm in Figure 4.8 where we use the auxiliary function `treated` presented in Figure 4.9.

The algorithm visits all the elements of the trace, comparing each of them with the first component of the invariant. If the current element of the trace matches the input/output pair presented in the invariant, the algorithm checks if the associated time value falls in the interval marked in the invariant. If this holds, then the part of the invariant that has not been checked and the time registered in the current position of the trace are stored in a stack. In this way, we will have a buffer with all the *pending* situations that must be checked when the algorithm reaches the next position of the trace. Thus, for each step of the algorithm we will push a new element in the stack, if the new position reached in the trace fulfills the requirements of the invariant. In addition, we will check all the pending situations in the stack against the new element of the trace. If it does not hold, the element is removed from the stack. On the contrary, if it holds, then the pending situation is updated with the remaining part of the invariant and the time of the element in the trace. Let us remark that the fact that the algorithm finds no match of the recorded log with the invariant when we are checking the first $n-1$ elements of the invariant does not indicate that the trace does not fulfill the invariant. In that case, we have not found the preconditions established by it. It is only when we reach the last component of the invariant for each of the pending situations, when a verdict can be emitted. If we find an error then the algorithm stops; otherwise, it continues reviewing the rest of the trace and the elements remaining in the stack.

input : $s :: sequence$,
$I = \{a_1/\hat{p_1}, \ldots, a_{n-1}/\hat{p_{n-1}}, i_n \mapsto O/\hat{p_n} \rhd \hat{p}\}$
// *where for all* $1 \leq k \leq n-1$ *we have that* $\hat{p}_k \in \mathcal{IR}$,
// *and either* $a_k = i_k/o_k$, *with* $i_k \in \mathcal{I} \cup \{?\}$ *and* $o_k \in \mathcal{O} \cup \{?\}$, *or* $a_k = \star$;
// $i_n \in \mathcal{I}$, $O \subseteq \mathcal{O}$, *and* $\hat{p_n}, \hat{p} \in \mathcal{IR}$.
output: `Bool`.

Struct $\mathcal{A}$ { $t_t :: \mathbb{R}_+$;
$t_e :: \mathcal{IR}$;
$t_a :: \mathbb{R}_+$;
$wild ::$ `Bool`;
$I_{aux} ::$ FixedTimeInv}
$b :: Stack[\mathcal{A}]$;
$b_{aux} :: Stack[\mathcal{A}]$;
$token :: \mathcal{A}$;
$error \leftarrow$ `false`;
$j \leftarrow 1$;

while $(j \neq$ `length`$(s) \ \wedge \neg error)$ **do**
  (i / o / t)←s[j];
  $j \leftarrow j + 1$;
  $token.t_t \leftarrow 0$;
  $token.t_e \leftarrow [0, 0]$;
  $token.t_a \leftarrow 0$;
  $token.wild \leftarrow$ `false`;
  $token.I_{aux} \leftarrow I$;
  $aux \leftarrow$ `treated`$((i \ / \ o \ / \ t), token, error)$;
  if $(aux \neq null)$ **then**
    `push`(b_{aux}, aux);
  while $\neg$(`isEmpty`(b)) **do**
    $token \leftarrow$ `top`(b);
    $aux \leftarrow$ `treated`$((i \ / \ o \ / \ t), token, error)$;
    if $(aux \neq null)$ **then**
      `push`(b_{aux}, aux);
  $b \leftarrow b_{aux}$;
return$(\neg error)$;

Figure 4.8: Correctness of a log with respect to an invariant in FixedTimeInv.

```
input  : (i / o / t),
         token::A,
         &error :: Bool.
output: A.
switch (head(token.I_aux)) do
    Case : (i_m / o_m/ p̂_m)
    if ((i = i_m) then
        if [(token.wild ∧ token.t_a ∈ token.t_e) ∨ ¬ token.wild] ∧ o = o_m ∧ t ∈ p̂_m then
            token.t_t ← token.t_t + t;
            token.t_e ← [0, 0]; token.t_a ← 0; token.wild ← false;
            token.I_aux ← tail(token.I_aux); return(token);
        else
            return(null);
    else
        token.t_a ← token.t_a + t;
        if (token.wild ∧ (t_e ⊙ (token.t_a)) then
            token.t_t ← token.t_t + t; token.I_aux ← tail(token.I_aux); return(token);
        else
            return(null);
    Case :(i_n ↦ O/p̂_n ▷ p̂)
    if (i = i_n) ∧ ((¬token.wild) ∨ (token.wild ∧ (t_e ⊙ (token.t_a + t)))) then
        token.t_t ← token.t_t + t;
        if ((o ∈ O) ∧ (t ∈ p̂_n) ∧ (token.t_t ∈ p̂)) then
            return(null);
        else
            error ← true; return(null);
    else
        return(null);
    Case :(⋆_m, p̂_m)
        token.t_t ← token.t_t + t;    token.t_e ← p̂_m;    token.t_a ← t;
        token.wild ← true;    token.I_aux ← tail(token.I_aux);    return(token);
```

Figure 4.9: `treated` function.

The function `treated` checks if an element of the trace and a component of the invariant match. In this function, the treatment is different depending on the kind of component of the invariant being checked. The first one corresponds to elements of the form (input/output/time); the second one deals with the very last part of the invariant. Finally, the third one manages those elements that contain a $\star$ symbol. Let us remark that in the second case we are at the end of the invariant and we have to check all the restrictions imposed by it. It is the only place of the function where an error can be found.

Regarding the complexity of our pattern matching strategy, in the worst case we obtain $\mathcal{O}(m \cdot n)$ (n is the length of the invariant and m is the length of the observed trace). Let us remark that even though *good* algorithms for pattern matching on strings perform in $\mathcal{O}(m)$ (after the *pre-processing* phase) we cannot achieve this complexity because we must check all the occurrences of the pattern in the trace. However, as we commented before, if we consider that the length of the invariant is *much smaller* than the length of the trace, as it is usually the case, we have that this complexity is almost linear with respect to the length of the trace.

4.3 Stochastic Time Approach

Next we present the second approach for making passive testing using stochastic information. The rest of this section is organized follows. We start this section with a small motivation about the use of stochastic information in computational systems. Next, we will propose our framework based on the use of invariants in environments with stochastic information. After that, following the schema used in the fix time approach invariants, we will provide two algorithms: One of them will be used to decide the correctness of an invariant with respect a specification and the other one to decide the correctness with respect to a trace.

In probability theory, a *stochastic process* is the counterpart to a deterministic process. Instead of dealing with only one possible *reality* of how the process might evolve under time, in a stochastic or random process there is some indeterminacy in its future evolution described by probability distributions. This means that even if the initial condition (or starting point) is known, there are many possibilities the process might go to, but some paths are more probable and others are less. A basic type of stochastic process is the one that can amount to a sequence of random variables known as a time series (for example Markov chain).

In this Master Thesis we propose that it can be possible that in a specification, the time

can be represented with stochastic information. In order to describe time properties with stochastic information associated with time, we introduce *stochastic time invariants*. The invariants are similar to fix time invariants, but their behaviour is not equivalent. With stochastic time invariants we are able to express new situations such as

> "*After pressing the red button we receive a coke before* 4 *seconds with probability 0.95*".

Definition 4.8 Let $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$ be a $\texttt{TFSM}_{st}$. We say that the sequence I is a *stochastic time invariant* for M if the following two conditions hold:

1. I is defined according to the following EBNF:

$$I ::= a/z/F, I \mid \star, I' \mid i \mapsto O/\mathcal{G}$$
$$I' ::= i/z/F, I \mid i \mapsto O/\mathcal{G}$$

   In this expression we consider $F \in \mathcal{F}$, $i \in \mathcal{I}$, $a \in \mathcal{I} \cup \{?\}$, $z \in \mathcal{O} \cup \{?\}$, $\mathcal{G} \subseteq \mathcal{F}$, and $O \subseteq \mathcal{O}$.

2. I is *correct* with respect to M.

We denote the set of stochastic time invariants by StochasticTimeInv. □

Let us remark that, in this setting, invariants do not check the total time. Alternatively, we could have considered an interval to do this task, but let us note that a probability distribution function could not play this role. In order to describe more real systems we assume that implementations have *regular stochastic information*. Let us show some usual situations to illustrate this assumption. Let us suppose that we are implementing a software program P. In P we define a function having only one input parameter, belonging to $\mathbb{N}$, and returning a result in `Bool`. It is a good assumption to assume that in all states of the specification where the implementer needs to perform an action similar to P she will call P. Let us suppose that this function has associated a probability distribution function F to compute the amount of time that the computation of the boolean takes. If there is an error associated with the implementation of this probability distribution function, such as it is wrongly implemented, or the dependency modules perform bad requests, will be produced a different function F' associated with this transition. This means that all comparisons relative to this function will not behave correctly.

Example 4.4 Next we illustrate the idea of stochastic invariant and give some examples to illustrate their behaviour. Let us consider the following invariant $i \mapsto \{o\}/\{F\}$. The idea is

that each occurrence of the symbol i is followed by the output symbol o and this transition is performed in an amount of time that can be generated by the function F.

We can specify a more complex property by taking into account that we are interested in observing the outputs o_1 or o_2 after the input i only if the input i_0 was previously observed. In addition, we want to indicate that always that i_0 is observed *in any* part of the trace, then the amount of time to receive any request is generated by F_0. Furthermore, we can express that along the length of the trace, always that we find i and it is follows by o_1 or o_2 the function associated with them are F or F'. This last property denotes for some flexibility about the trace. We could express this property by means of the invariant $i_0/?/F_0, \star, i \mapsto \{o_1, o_2\}/\{F_n, F'_n\}$. An observed trace will be correct with respect to this invariant if each time that we find a (sub)sequence starting with the input i_0 paired with any output symbol, then if there is an occurrence of any trace of inputs/outputs without showing the input=i, then when we obtain the input symbol i then it must be paired with the output symbol o_1 or o_2. In addition we collect all time values of any occurrence of i_0 followed by any output, and with i followed by o_1 or o_2. An observed set of recorder time observation will be correct with respect the invariant, by using the chi-square goodness test, these sets can be generated from F_0 and from F or F' respectively.

We can refine the previous invariant if we consider only the cases where the pair i_0/o_0 was observed. The invariant for denoting this property is $i_0/o_0/F_0, \star, i \mapsto \{o\}/\{F, F'\}$. Let us remark that we could not deduce that we have found an error if the pair i_0/o_0 appears in the observed trace but the input i is not detected afterwards in the corresponding trace. In such a situation we cannot conclude that the implementation fails. Similarly, if we find the pair i_0/o_1 we cannot conclude anything since the premise of the invariant, that is, the whole sequence but the last pair was not found. An invariant as $i \mapsto \{o_1, o_2\}/\{F, F'\}$ indicates that after input i we observe either the output o_1 or o_2; in addition, if we collect together all the time values associated with the performance of i/o_1 and i/o_2 then this sample fits either F or F'. □

Definition 4.9 Let $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$ be a $\texttt{TFSM}_{ft}$, $s \in S$, $a \in \mathcal{I} \cup \{?\}$, $z \in \mathcal{O} \cup \{?\}$. We define the set $\texttt{afterCond}(s, i, z)$ as the set of transitions belonging to Tr having as initial state s and performing the input i before the output z.

$$\begin{aligned}
\texttt{afterCond}(s,i,o) &= \{(s,s',i,o,F)|(s,s',i,o,F) \in Tr\} \\
\texttt{afterCond}(s,?,o) &= \bigcup_{i\in\mathcal{I}} \texttt{afterCond}(s,i,o) \\
\texttt{afterCond}(s,i,?) &= \bigcup_{o\in\mathcal{O}} \texttt{afterCond}(s,i,o) \\
\texttt{afterCond}(s,?,?) &= \bigcup_{i\in\mathcal{I},o\in\mathcal{O}} \texttt{afterCond}(s,i,o)
\end{aligned}$$

We define the function $\texttt{afterInp}(s,i)$ as the function that computes the set of states that can be reached from state s without performing the input i. Formally we define $\texttt{afterInp}(s,i)$ as:

$$\texttt{afterInp}(s,i) = \{s'|(s,s',i',o,F) \in Tr \wedge i \neq i'\}$$

□

Next we present the algorithm to determine the correctness of an invariant in StochasticTimeInv with respect a specification. This algorithm is described in Figure 4.10 and in Figure 4.11. In this algorithm we distinguish two different parts. In the first part of the algorithm, see Figure 4.10 we generate all possible set of states where the invariant can be started. After that, it applies a loop along the length of the invariant to determine the set of possible states at the end (once the head of the invariant is reached). In the first part of the algorithm we return false if there is no state at the end of the loop, meaning that the invariant is useless for this specification.

In the second part of the algorithm, we determine whether there is an error due to outputs or the probability functions error. We have a set of reached states, and we are in the head of the invariant. Let us remember that the head of the invariant is $i \mapsto O/\mathcal{G}$. Now the algorithm determines that after the last input of the invariant, that is i, we have an output in O and a probability distribution function in G. If any of the restrictions do not hold, then we return that the invariant is not correct with respect to the specification.

Lemma 4.2 Let $M = (S,\mathcal{I},\mathcal{O},Tr,s_{in})$ be a $\texttt{TFSM}_{st}$. The worst case of the algorithm given in Figure 4.10 and in Figure 4.11 checks the correctness of an invariant belonging to StochasticTimeInv $I = i_1/o_1/F_1,\ldots,i_{n-1}/o_{n-1}/F_{n-1},i_n \mapsto O/\mathcal{G}$ with respect to M:

- In time $\mathcal{O}(n\cdot|Tr|)$ and space $\mathcal{O}(|Tr|)$ if I does not present occurrences of $\star$.
- In time $\mathcal{O}(k\cdot|Tr|^2+(n-k)\cdot|Tr|)$ and space $\mathcal{O}(|Tr|)$ if I presents k occurrences of $\star$ in I.

□

```
in  : M = (S, I, O, Tr, s_in)
      // M is a TFSM_st
      I = {a_1, ..., a_{n-1}, i_n ↦ O/G}
      // and either a_k = i_k/o_k/F_k, with i_k ∈ I ∪ {?}, o_k ∈ O ∪ {?}
      // and F_k ∈ F, or a_k = ⋆;  i_n ∈ I, O ⊆ O, and G ⊆ F.
out: Bool.
I' = I; S' ← S; j ← 1; S'' ← ∅;
while (j < n) do
    if (head(I') = ⋆) then
        while (S' ≠ ∅) do
            Choose s ∈ S';
            S' ← S' \ {s};
            S'' ← S'' ∪ afterInp(s, i_{j+1});
    else
        while (S' ≠ ∅) do
            Choose s_a ∈ S';
            S' ← S' \ {s_a};
            Tr' ← afterCond(s_a, i_j, o_j);
            while (Tr' ≠ ∅) do
                Choose (s_a, s_b, i_j, o_j, F') ∈ Tr';
                Tr' ← Tr' \ {(s_a, s_b, i_j, o_j, F')};
                if F' = F_j then
                    S'' ← S'' ∪ {s_b};
    I' = tail(I');
    j ← j + 1;  S' ← S''; S'' ← ∅;
error ← false;
if (S' = ∅) then
    error ← true;
```

Figure 4.10: Correctness of an invariant in StochasticTimeInv with respect to a specification (1/2).

```
while (S' ≠ ∅) do
    Choose s_a ∈ S';
    S' ← S' \ {s_a};
    Tr' ← afterCond(s_a, i_n, ?);
    while (Tr' ≠ ∅) do
        Choose (s_a, s_b, i_n, o, F') ∈ Tr';
        Tr' ← Tr' \ {(s_a, s_b, i_n, o, F')};
        if ((o ∈ O) ∧ (F' ∈ G)) then
            S'' ← S'' ∪ {s_b};
        else
            error ← true
if (S'' = ∅) then
    error ← true;
Return (¬error);
```

Figure 4.11: Correctness of an invariant in StochasticTimeInv with respect to a specification (2/2).

Conformance of traces with respect to invariants

Next, we describe how can we use stochastic time invariants in order to detect wrong behavior of an `IUT`. First, we need to obtain a log from the `IUT`. Let us remember that a log is a recorder trace of all observable interactions with the implementation. For making it, we consider the inputs, the outputs and the time the system takes to perform the output.

In Figure 4.12 we describe the algorithm that we use to establish the conformance of a trace obtained from the `IUT` with respect to an invariant. We present the core of the algorithm in Figure 4.12, where we use the auxiliary function `treated` presented in Figure 4.13.

The algorithm for checking the conformance of a trace has two difference stages. The first one includes the correction with respect the input/output pairs while the second one is relative to the time restrictions expressed in the invariant.

In the first stage we run along the trace, looking for any errorneus behaviour expressed in the invariant. For having a good performance in this task, we only traverse the trace once. To help in this idea we use a stack where we are saving and updating data with regarding the explorer section of the log. The function `treated` checks whether an element of the trace and a component of the invariant match. In this function, the treatment is different depending on the kind of component of the invariant being checked. The first one

input : $s :: sequence$
$I = \{a_1, \ldots, a_{n-1}, i_n \mapsto O/\mathcal{G}\}$
$\delta \in [0,1]$level of confidence
output: `Bool`.

$times :: \wp(\mathcal{T}) :: [|I| \times |O|]$;
//Set of multisets of $\mathcal{T}$ with length $|I| \times |O|$
Struct $\mathcal{A}$ { $wild ::$ `Bool`;
$I_{aux} ::$ StochasticTimeInv}
$b :: Stack[\mathcal{A}]$; $b_{aux} :: Stack[\mathcal{A}]$; $token :: \mathcal{A}$; $j \leftarrow 1$; $error \leftarrow$ `false`; $I' \leftarrow I$;

while $(j \neq$ `length`$(s) \ \wedge \neg error)$ **do**
  (i / o / t)$\leftarrow$s[j]; $j \leftarrow j+1$; $times_{io} \leftarrow times_{io} \cup \{t\}$;
  $token.wild \leftarrow$ `false`; $token.I_{aux} \leftarrow I$; $aux \leftarrow$ `treated`$((i \ / \ o \ / \ t), token, error)$;
  if $(aux \neq null)$ **then**
    `push`(b_{aux}, aux);
  while $\neg$(`isEmpty`(b)) **do**
    $token \leftarrow$ `top`(b); $aux \leftarrow$ `treated`$((i \ / \ o \ / \ t), token, error)$;
    if $(aux \neq null)$ **then**
      `push`(b_{aux}, aux);
  $b \leftarrow b_{aux}$;
while $(I' \neq \ \wedge \neg error)$ **do**
  if (`head`$(I') = (i \ / \ o \ / \ F)$) **then**
    if $(\gamma(F, times_{io}) < \delta)$ **then**
      $error \leftarrow$ `true` ;
  else
    if (`head`$(I') = (i \ / \ O \ / \ \mathcal{G})$) **then**
      $find \leftarrow$ `false`;
      while $(O \neq \emptyset \wedge \neg find)$ **do**
        `Choose` $o \in O$; $O \leftarrow O \setminus \{o\}$; $G' \leftarrow G$;
        while $(G' \neq \emptyset \wedge \neg find)$ **do**
          `Choose` $F \in G'$; $G' \leftarrow G' \setminus \{F\}$;
          if $(\gamma(F, times_{io}) \geq \delta)$ **then**
            $find \leftarrow$ `true`;
      if $(\neg find)$ **then**
        $error \leftarrow$ `true`;
  $I' \leftarrow$ `tail`(I');
return$(\neg error)$;

Figure 4.12: Correctness of a log with respect to an invariant in StochasticTimeInv.

```
input  : (i / o / t)
         token::A
         &error :: Bool
output: A.
switch (head(token.I_aux)) do
    Case : (i_m / o_m/ t_m)
    if ((i = i_m) then
        if o = o_m then
            token.wild ← false; token.I_aux ← tail(token.I_aux);
            return(token);
        else
            return(null);
    else
        if (token.wild) then
            return(token);
        else
            return(null);
    Case :(i_n ↦ O/G)
    if (i = i_n) then
        if ((o ∈ O)) then
            return(null);
        else
            error ← true;
            return(null);
    else
        return(null);
    Case :(⋆)
        token.wild ← true;   token.I_aux ← tail(token.I_aux);   return(token);
```

Figure 4.13: `treated` function.

corresponds to elements of the form input/output/time while the second one deals with the very last part of the invariant. Finally, the third one manages those elements that contain a $\star$ symbol. Let us remark that in the second case we are at the end of the invariant and we have to check the output restrictions imposed by it. It is the only place of the function where an error can be found.

After that, we start with the second stage. For this task we would need to go along the trace, saving the time values associated with this input/output. But, in order to improve the performance of the algorithm we include this task in the previous stage. The data structure that we use for saving all those values is a set of multisets. After having all those stored values, we need to check all time restrictions imposed by the invariant. We need to go along this, in order to check the correctness of time values. We use the input parameter λ to denote the minimum confidence that we can let to the data of the trace. An error happens if any sample associated with a time restriction does not pass the fitness test.

Regarding the complexity of our pattern matching strategy, in the worst case we obtain $\mathcal{O}(m \cdot n)$ (n is the length of the invariant and m is the length of the observed trace) again.

Chapter 5

PasTe: a PASssive TEsting tool

In addition to the theoretical framework we have developed a tool called PasTe that helps in the automation of our passive testing approaches. In particular, all algorithms presented in this Master Thesis are fully implemented. Throughout this chapter we will show some of the most relevant aspects of PasTe and we will analyze some experiments and their results. Next we briefly explain how this chapter is organized.

In Section 5.1 we will go into details of the (input) data representation, by using the Extensible Markup Language (XML), in PasTe. First, we will specify how specifications must be represented both by using $\texttt{TFSM}_{ft}$ and $\texttt{TFSM}_{st}$ models. Second, we will describe how traces from an `IUT` are represented, or if we are not provided with them we will show how PasTe can automatically generate them from a machine representing a possible `IUT`. After that, we will describe how the the probability distribution functions are represented and to finish this section we will show the invariants representation.

As we have just mentioned, PasTe allows two different ways to introduce traces. On the one hand testers can have a set of traces produced by an `IUT` and introduce them into the system by using the XML format. On the other hand, in some situations, we do not have these traces and we need to know how good a set of defined invariants with respect to a set of traces is. In order to overcome them the lack of traces, PasTe implements a based-mutant-specification approach for generating them. In Section 5.2 we will show details of our approach.

Next, in Section 5.3 we will present the internal core of PasTe and we will mention the most relevant conformance functions and algorithms included in it.

In order to conclude this chapter, in Section 5.4 we will show some experiments performed with PasTe and we will present some relevant empirical results obtained from the evaluation

of the two methodologies described in this Master Thesis.

5.1 Representation of input data

Next we present the structure of the main XML TAGS that we use within our tool.

1. `Specification by using TFSM formalism.`

   - SPECIFICATION: This tag is the first one in order to describe a specification.
   - TYPE: Represents the nature of the specification. The permitted values are: FIXED_TIME ($\texttt{TFSM}_{ft}$) and STOCHASTIC_TIME ($\texttt{TFSM}_{st}$).
   - INITIAL_STATES: Set of initial states of the `TFSM`.
   - TRANSITIONS: Set of transitions.
   - TRANSITION: Tag for the transition.
   - INPUT: Tag for input.
   - OUTPUT: Tag for output.
   - TIME: Tag for fixed time value.
   - STATE_I: Tag for initial state.
   - STATE_F: Tag for final state.
   - FUNCTION_NICK: Tag for the *Nick* of a probability distribution function. The definition of the current function is provided in the function definition section of the XMLdocument.

   Next we show how we could represent, by using these tags, the specifications given in 4.2. In Figure 5.1 we provide the XML that represents two transition with fixed time values, which correspond with the $t_{22} = s_2 \xrightarrow{i_0/o_0}_4 s_2$ and $t_{12} = s_1 \xrightarrow{i_0/o_1}_4 s_2$ respectively, extracted from the Figure 4.2.

2. `Probability distribution functions.`

   - FUNCTIONS: Tag for the set of probability distribution function.
   - FUNCTION: Tag for a probability distribution function.
   - NAME: Tag for representing the nature of the function. Currently allowed values are UNIFORM, DISCRETE, BINOMIAL, DIRAC, EXPONENTIAL.

```
<SPECIFICATION>
        <TYPE>FIX-TIME</TYPE>
        <INITIAL_STATES>
                <STATE>s1</STATE>
        </INITIAL_STATES>
        <TRANSTIONS>
                <TRANSITION>
                  <STATE-I>s2</STATE-I>
                  <STATE-F>s2</STATE-F>
                  <INPUT>i0</INPUT>
                  <OUTPUT>o0</OUTPUT>
                  <TIME>4.0</TIME>
                </TRANSITION>
                <TRANSITION>
                  <STATE-I>s1</STATE-I>
                  <STATE-F>s2</STATE-F>
                  <INPUT>i0</INPUT>
                  <OUTPUT>o1</OUTPUT>
                  <TIME>4.0</TIME>
                </TRANSITION>
        </TRANSITIONS>
</SPECIFICATION>
```

Figure 5.1: Representation of two transitions t_{22} and t_{12} from Figure 4.2 in XML format.

  - UNIFORM: It uses two parameters to denote the interval $[\alpha, \beta]$. Tags names are ALFA and BETA, respectively.
  - DISCRETE: It uses $2 \cdot n$ parameters, being n the number of elements that are needed for defining the function. We provide a tuple conformed by a value v and the probability associated with it p. In this case, the tags are: PAIRS for denoting the set of pairs, PAIR for denoting a pair $<v,p>$, VALUE to denote v and PROBABILITY that corresponds to p.
  - BINOMIAL: The two parameters needed for defining a binomial distribution are p and n. The tags are P and N.
  - DIRAC: This is a particular case of discrete probability distribution function. Only one parameter is needed in the DIRAC tag, the only value with probability 1. The tag for this value is VALUE.
  - EXPONENTIAL: For defining the exponential function only one parameter must be provided. The tag is VALUE.

- NICK: The Nick representation for this function. This tag will be used in transition and in invariant definitions.

Next in Figure 5.2 we show the same process for the $t_{34} = s_3 \xrightarrow{i_2/o_3}_{F_2} s_4$ from the $\texttt{TFSM}_{ft}$ defined in Figure 4.3. Let us note that we need first to define the function F_2 in it. F_2 was defined as

```
<FUNCTIONS>
        <FUNCTION>
                <NAME> DIRAC </NAME>
                <VALUE> 4.0 </VALUE>
                <NICK> F2 </NICK>
        </FUNCTION>
</FUNCTIONS>
<SPECIFICATION>
<TYPE>STOCHASTIC_TIME</TYPE>
      <INITIAL_STATES>
      <STATE>s1</STATE>
      </INITIAL_STATES>
      <TRANSITIONS>
              <TRANSITION>
                      <STATE_I>s3</STATE_I>
                      <STATE_F>s4</STATE_F>
                      <INPUT>i1</INPUT>
                      <OUTPUT>o3</OUTPUT>
                      <FUNCTION_NICK>F2</FUNCTION_NICK>
              </TRANSITION>
      </TRANSITIONS>
</SPECIFICATION>
```

Figure 5.2: Representation of transitions t_{34} from Figure 4.3 in XML format.

$$F_2(t) = \begin{cases} 0 & \text{if } t < 4 \\ 1 & \text{if } t \geq 4 \end{cases}$$

3. `Traces.` Let us remember that a trace is a sequence of terms of <input/output/time>.

   - TRACES: Set of traces.
   - TRACE: Tag for denoting a trace.
   - INPUT: Tag for input.
   - OUTPUT: Tag for output.
   - VALUE: Tag for timed values.

4. `Invariants.` We divide the description of an invariant in two parts: The tail and the head. Let us remember that, depending on the approach, the tail can be composed by sequences <input/output/interval> or <input/output/Function>. In the case of using wild-char-like ? or $\star$, they will be inserted in the input/output tags respectively. If either input or an output tag is $\star$ then this sequence will be consider like $\star$.

   - SEQUENCES: Set of sequences.
   - SEQUENCE: Tag for a sequence.

- INPUT: Tag for inputs.
- OUTPUT: Tag for outputs.
- INTERVAL: Tags for the interval $[\alpha, \beta]$. This tag will include two tags for the bounds: ALFA and BETA.
- FUNCTION_NICK: Nick associated with the function.
- HEAD: Tag for denoting the head of the invariant.
- SET_OUTPUTS: Set of outputs in the head of the invariant.
- SET_FUNCTIONS Set of probability distribution functions defined in the head of the invariant.
- LAST_INTERVAL Tags for denoting the last interval $[\alpha, \beta]$ of the invariant. We denote by ALFA and BETA the tags for the values α and β respectively.
- GENERAL_INTERVAL Tag for denoting the interval for the total time restriction of the invariant. ALFA and BETA are that tags that will be used to express it.

In Figure 5.3 we represent an invariant in FixedTimeInv. The represented invariant is $I = i_2/o_1[10, 13], i_2 \mapsto \{o_2, o_3, o_4\}[7, 20] \rhd [14, 40]$.

Let $I = i_1 \mapsto \{o_1, o_2, o_4\}\{F\}$ be a StochasticTimeInv with

$$F(t) = \begin{cases} 0 & \text{if } t < 1 \\ 0.1 & \text{if } 1 \leq t < 2 \\ 0.6 & \text{if } 2 \leq t < 3 \\ 1 & \text{if } 3 \leq t \end{cases}$$

We show the representation of I in XML in Figure 5.4.

5.2 Acquiring implementations

Let us suppose that we are provided with a specification and a set of invariants, but we do not have real traces from an `IUT`. PasTe provides a technique to automatically generate traces from an `IUT`represented as a `TFSM`. This approach is based on *mutants*. With a mutant we would like to simulate a possible implementation. In particular, PasTe creates a set of mutants by making some small changes in the transitions of the specification. For example these changes can be wrong outputs, changing the final state from a transition or providing a different associated time. In this last case, the change of time can be done in our first

```
<INVARIANTS>
      <INVARIANT>
      <SEQUENCE>
            <INPUT>i2</INPUT>
            <OUTPUT>o1</OUTPUT>
            <INTERVAL>
              <ALFA>10</ALFA>
              <BETA>13</BETA>
            </INTERVAL>
      </SEQUENCE>
      <HEAD>
            <INPUT>i2</INPUT>
            <SET_OUTPUTS>
              <OUTPUT>o2</OUTPUT>
              <OUTPUT>o3</OUTPUT>
              <OUTPUT>o4</OUTPUT>
              </SET_OUTPUTS>
            <LAST_INTERVAL>
              <ALFA>7</ALFA>
              <BETA>20</BETA>
            </LAST_INTERVAL>
            <GENERAL_INTERVAL>
              <ALFA>14</ALFA>
              <BETA>40</BETA>
            </GENERAL_INTERVAL>
      </HEAD>
      </INVARIANT>
</INVARIANTS>
```

Figure 5.3: Representation of an invariant in FixedTimeInv in XML format.

setting by changing the value associated to that transition, and in the second approach by changing the definition of the function.

We will use the following notation to denote the mutant operators: Let M be a specification, we say that M_o is an *output mutant* denoted by $M \rightsquigarrow_o M_o$, if M_o is equal to M but the change of an output of one transition is produced. We consider that M_s is a *state mutant* denoted by $M \rightsquigarrow_s M_s$ if M_s is constructed from M by changing the goal state of a transition. We consider that M_t is a time mutant denoted by $M \rightsquigarrow_t M_t$ if a change in one time parameter has been produced. The mutants will have an additional condition depends on the formalism that we are using. If we have $\texttt{TFSM}_{ft}$ then we just need to change a fix time value associated with a transition by another new value. However if we are using a $\texttt{TFSM}_{st}$ model then we have to change the definition of a function in $\mathcal{F}$ associated with a set of transitions.

Once we have a set of mutants, we generate random traces from these mutants to obtain a big sample where we are able to test the proposed invariants. Let us suppose that $M'_\alpha = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$, is a mutant. The loop for generating traces is described as follows:

1. We start in the initial state, $s' \leftarrow s_{in}$.

```
<FUNCTIONS>
        <FUNCTION>
                <NAME> DISCRETE </NAME>
                <PAIRS>
                    <PAIR><VALUE>1</VALUE><PROBABILITY>0.1</PROBABILITY></PAIR>
                    <PAIR><VALUE>2</VALUE><PROBABILITY>0.5</PROBABILITY></PAIR>
                    <PAIR><VALUE>3</VALUE><PROBABILITY>0.4</PROBABILITY></PAIR>
                </PAIRS>
                <NICK>F</NICK>
        </FUNCTION>
</FUNCTIONS>
<INVARIANTS>
<INVARIANT>
        <HEAD>
            <INPUT>i1</INPUT>
            <SET_OUTPUTS>
                    <OUTPUT>o1</OUTPUT>
                    <OUTPUT>o2</OUTPUT>
                    <OUTPUT>o4</OUTPUT>
            </SET_OUTPUTS>
            <SET_FUNCTIONS>
                    <FUNCTION>
                    <FUNCTION_NICK>F</FUNCTION_NICK>
                    </FUNCTION>
            </SET_FUNCTIONS>
        </HEAD>
</INVARIANT>
</INVARIANTS>
```

Figure 5.4: Representation of an invariant in StochasticTimeInv in XML format.

2. Then we calculate all possible inputs $I' \subseteq \mathcal{I}$ that can be applied in that state.

3. Randomly we choose an input, $i' \in I'$.

4. Then we calculate the set of transitions $Tr' \subseteq Tr$ such that $s' \xrightarrow{i'/o}_{\alpha} s$.

5. Randomly we choose a transition $s' \xrightarrow{i'/o}_{\alpha} s$ and PasTe performs it.

6. We change the goal state $s' \leftarrow s$ and we jump to step 2 as long as we want to increase the length of the trace3.

5.3 Core of PasTe

In this section we comment the core of the tool PasTe. Initially, we assume that we have the following input data to manage:

- A specification (including the probability distribution function definitions).
- A set of invariants.

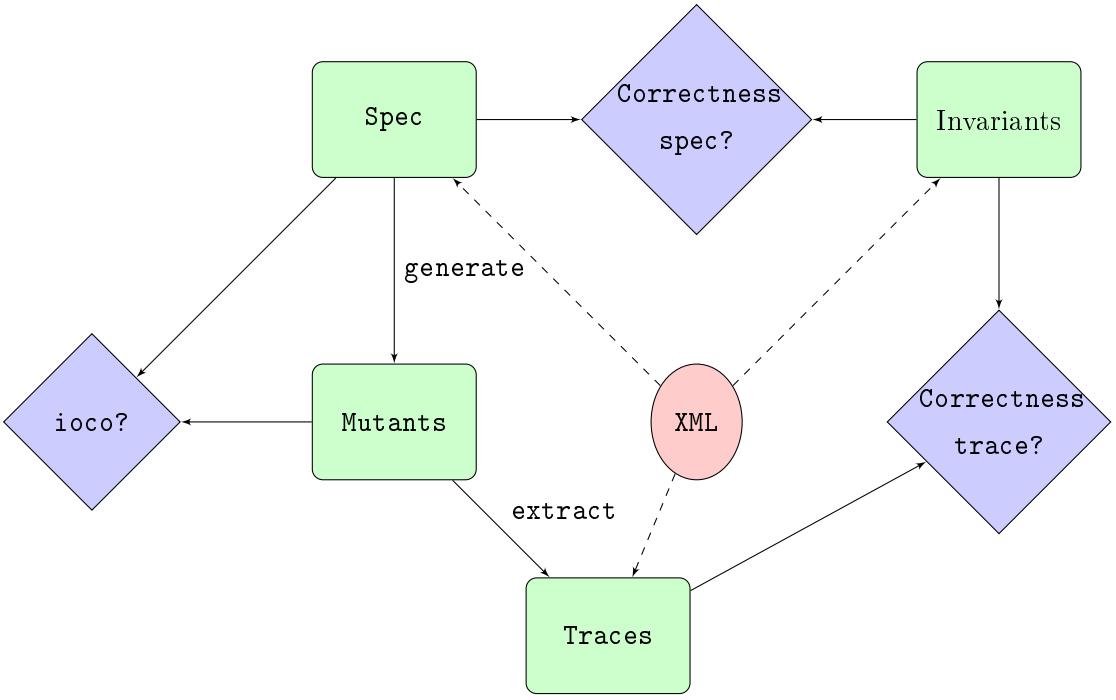


Figure 5.5: Core of decision algorithms of PasTe.

- Traces generated either by a `IUT` or by using mutants.

In Figure 5.5 we describe the scheme of PasTe Core. In the first place we have the node XML. This node represents the input file of our system. From this file we extract the specification and the invariants of a world. It can also contain traces from a `IUT`. If this traces are not in it, they will be generated by PasTe.

As we observe in the scheme, we have the block `Correctness spec?`. This blue box represents the two algorithms proposed in the previous chapter. These algorithms are used to check the correctness of the set of invariants (`Invariants` block) with respect the specification (`Spec` block). If an invariant is not correct then the set of all invariants is not correct.

If no traces are inserted from the XML file, then PasTe automatically generate mutants(`Mutants` block) from the specification. As we have seen, these mutants represent possible implementations extracted from the specification. Next, from this set of mutants PasTe extracts a set of traces (`Traces` block) which will be used to check the correctness with respect the invariants.

A new blue block, called `ioco`, is introduced in the core of PasTe. This algorithm is applied to a specification and a mutant. The idea is that we need to discard those mutants

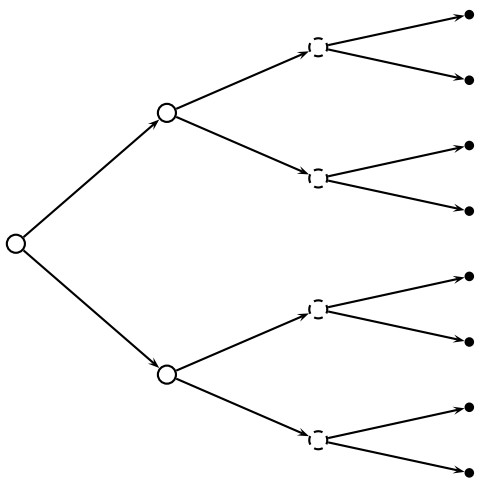


Figure 5.6: Specification with a multi-branch design.

that are not *real* mutants, that is, that they do not introduce a fault [Tre96, Tre99, Tre08]. This will be used to provide a way to decide if a mutant is conforming to a specification.

5.4 Results

In this section we present some results obtained from using PasTe. The goal of these experiments is to evaluate the performance of the methodologies proposed in this Master Thesis. First we will comment about a *classification* of possible specifications provided by testers. We will show that the performance in general of passive testing, and our methodology is in particular, strongly depends of the structure of the considered machines. Then, we will expose some results about `ioco` relationship. After that we will present and performance several experiments.

We start by describing our classification of specifications. On the one hand, we consider the class of specifications described in Figure 5.6. We will refer to this kind of specification as *tree-like*. This class can be used to specify systems deciding their behavour tree along. These systems usually do not have a way to return to a previous state. As we will observe during this section, this class is the one where our proposals badly perform. On the other hand, we have a class of specifications as the one represented in Figure 5.6. We will refer to this class as *connected*. Systems belonging to this class usually include ways to return to the initial state and to jump from one branch to another one.

As we commented in the previous section, we have implemented the `ioco` conformance

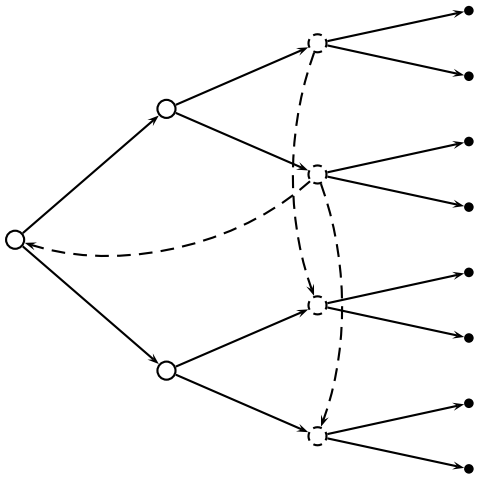


Figure 5.7: Specification with a connected design.

relation between mutants and the specification. The first experiment reported in Figure 5.8 consists in generate a set of mutants from a specification and observe the percentage to errors found by a set of invariants. In principle these mutants can or cannot be `ioco` conformance with respect the specification.

If we do not remove conforming mutants, the proportion of detected errors is 65.66%. The second value that we obtain is computed by removing this subset of mutants from the initial set. Then the proportion of detected errors increases to 73.50%. This experiment has been performed fifty times, by using a *connected* specification. In each simulation we use 25 mutants of each type. The number of traces generated are ten for each length. We consider the length of the trace taking into account the number of transitions of the mutant. For example, if the mutant has 25 transitions, then the trace $2x$ means that the length of the trace is 50. In these simulations we use traces of $1x$, $2x$, $3x$, $4x$, $5x$, $6x$, $7x$, $8x$, $9x$ length.

During the rest of experiments we will not remove conforming mutants. The reason is that we are working within a black box, so we are not able to detect if the mutant is `ioco` conforming or not.

We consider two different specifications. One of this specifications bellows to the tree-like class and the other one to the connected one. Since we are not provided with a set of traces, we let PasTe to generate mutants in order to perform a credible set of them. We generate 20 different mutants of each kind. Each mutant generates 10 traces with different lengths. In addition we are provided with two sets of corrects invariants. Each set contains

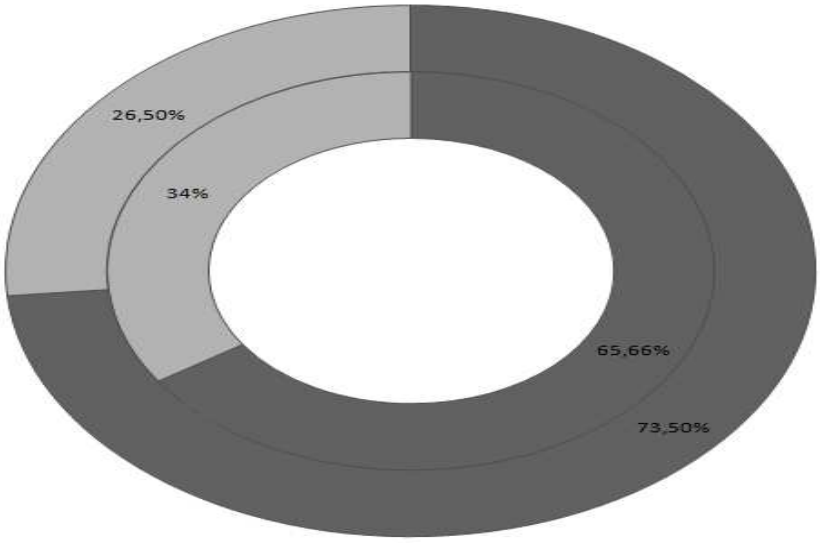


Figure 5.8: Proportion of errors found with/without removing conforming mutants .

40 invariants.

In Figures 5.9 and 5.10 we show the proportion of traces that have been detected as having errors from the set of all traces by each individual invariant. Let us note that the information in this figures does not reflect the probability of killing a mutant but the probability of finding an error in a given trace. For example Figure 5.9 we cannot infer that invariant 31 can find more error than 27. The experiment shows which invariant find more wrong traces but, each mutant generates 10 traces. So if invariant 31 can the 13% of the studied traces the max bound of error mutants that can find is 100%, he would find one produced wrong trace from each mutant, but the lower bound is 15%, it means all wrong detected traces are from 3 mutants. On the other hand, with 27 that can with the 7.72% of the studied traces, we have that the lower bound is 10% but the upper bound may be 75% too. For example, following the same reasoning, in Figure 5.10 we observe that invariants 29, 11, 16 can have the same power of mutant error detection.

For the next stage of the experiment, we will extract the more powerful invariants from the sets that we had. These invariants are, on the one hand $\{31, 37, 11, 1, 5, 10, 13, 19, 26, 30\}$ and on the other hand $\{29, 11, 6, 8, 34, 28, 35, 7, 32, 31\}$. For each specification, we also will produce 25 new different mutants of each class, and each mutant will produce 10 traces of each length. The possible lengths in this experiment are $1x$, $2x$, $3x$, $4x$, $5x$, $6x$, $7x$, $8x$,

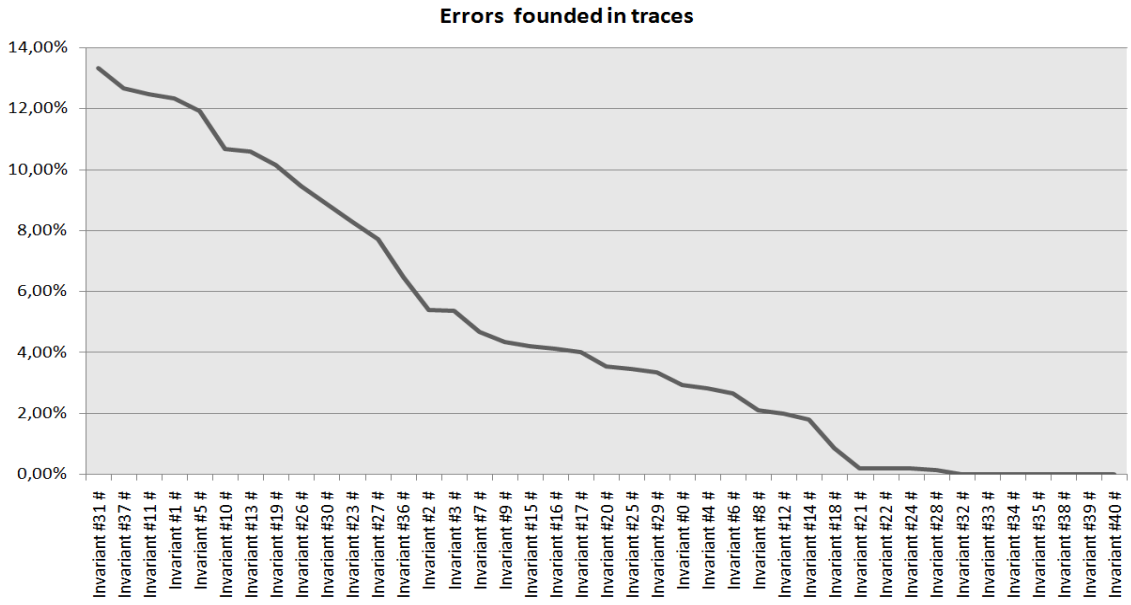


Figure 5.9: Proportion of erroneous traces detected in a generic connected specification by a set of 40 invariants.

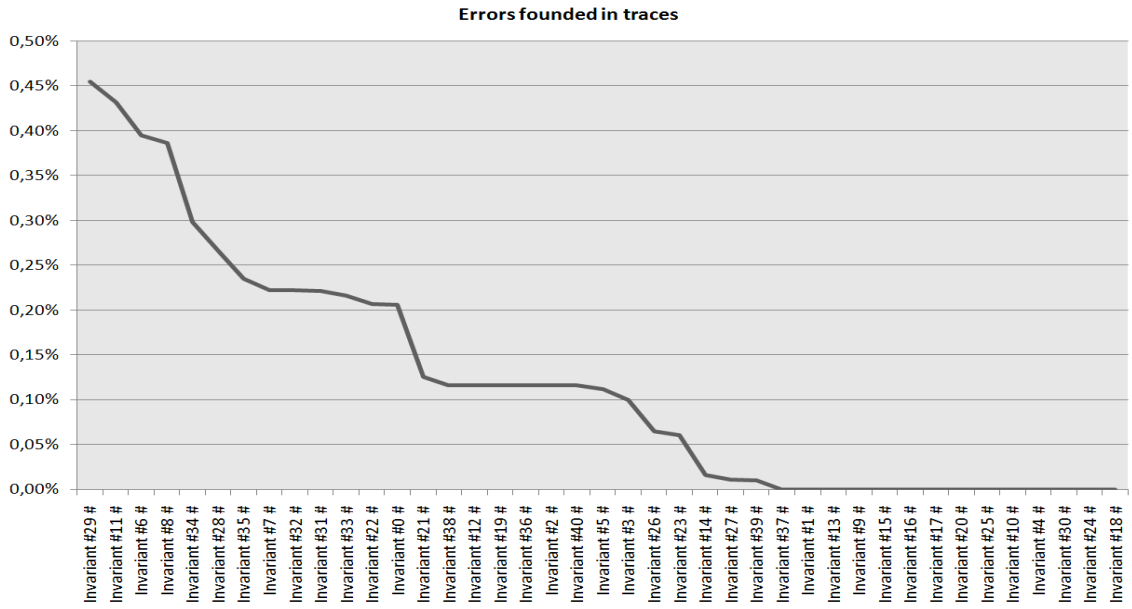


Figure 5.10: Proportion of erroneous traces detected in a generic tree-like specification by a set of 40 invariants.

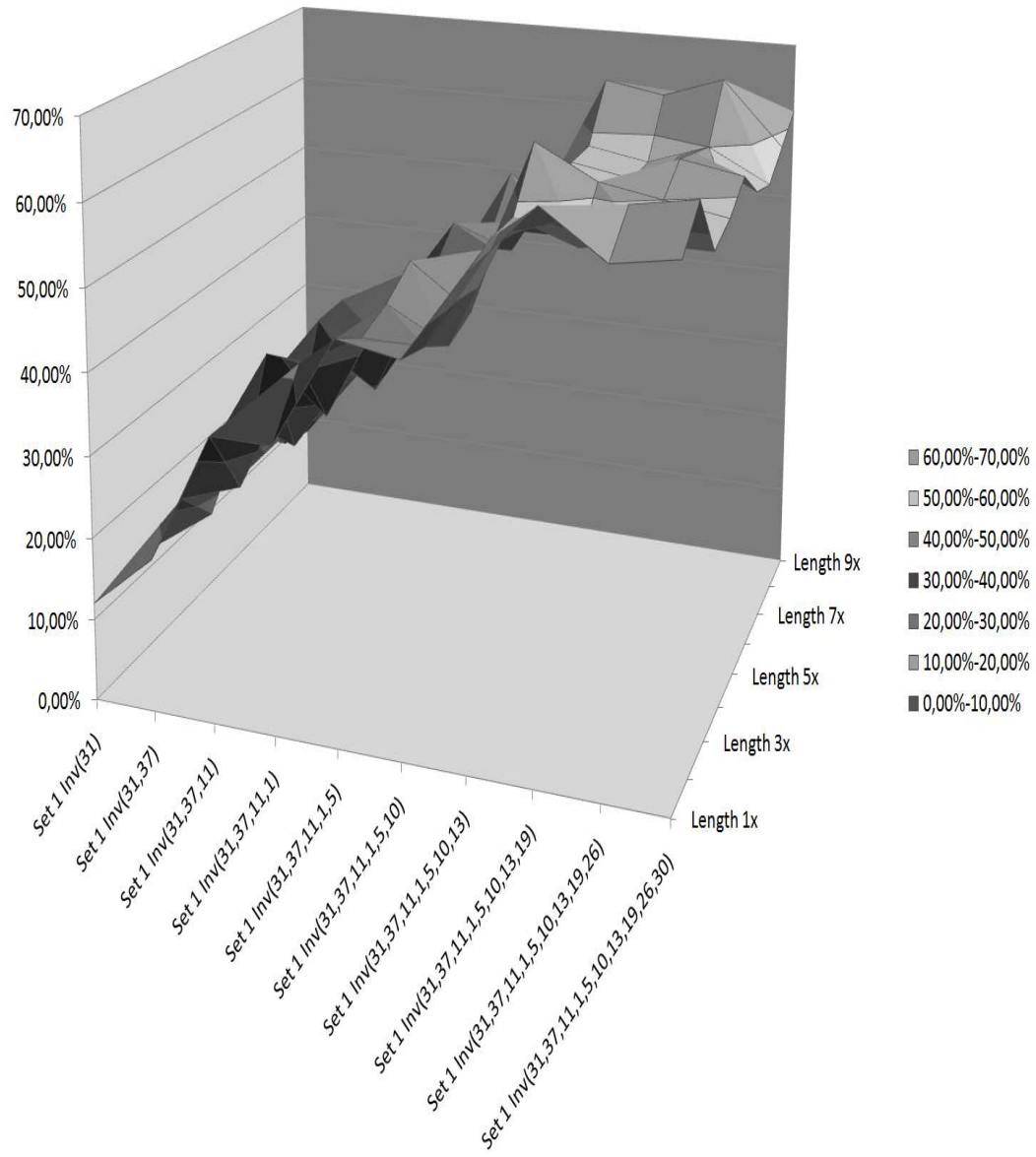


Figure 5.11: Relation between invariants, length of the log, and proportion of errors detected in a connected specification.

and $9x$, being x the number of transitions in our specifications: 25 transitions in each set of transitions. In Figure 5.11 we observe the obtained results from the specification belonging to the connected class. We stress that invariants are a very powerful tool to find errors among faulty implementations belonging to this class. Some of them can detect almost 10% of wrong mutants (see, numbers 31 and 37). Let us note that the percentage changes with respect the previous experiment because we have generated a bigger number of mutants. In Figure 5.11 we observe that the coverage of finding an error with a big set of invariants in this concrete specification is closer to 65%. Another result is direct and extracted that the power to detect faulty implementations depends on the length of the trace.

In Figure 5.12 we perform the experiment with the same setting as in the previous stage but changing the specification. In this case, we consider a *connected* specification while the set of the invariants is the one used to generate Figure 5.10. In this class of specifications the power of our approach is much lower than in the previous. The reason is that invariants only detect a subset of all traces that can be generated by the specification because one a path is chosen, it is impossible that the machines goes back to a previous state.

The last experiment is about the confidence that we can guarantee, in our approach when time is considered stochastic, with respect to length of the traces. This experiment has been performed by using only a class of mutants: Time mutants. The idea of this experiment is to deduce the degree of confidence that we can ask for a given trace. Let us note that low values of length in trace make the chi square goodness of fit test not to work properly. In Figure 5.13 we can observe the results. The values approximately follow a logarithm function. The differences between them are produced by the generator of numbers from the probability function. These values are the average of more than 1000 traces of lengths in {1x, 2x, 3x, 4x, 5x, 6x, 7x, 8x, 9x, 10x, 11x, 12x, 13x, 14x, 15x, 16x, 17x, 18x, 19x, 20x} being $x = 25$.

To summarize, in this chapter we have presented PasTe. PasTe helps us in the evaluation of the two approaches presented in this Master Thesis. We have presented the input data formalism for PasTe and the mutant approach in order to obtain possible traces from a specification. Finally we present some interesting experiments concluding that the approach has a good performance for making temporal passive testing what we call specifications *connected*. The complete API of PasTe can be founded in *http://kimba.mat.ucm.es/cesar/paste/api/*.

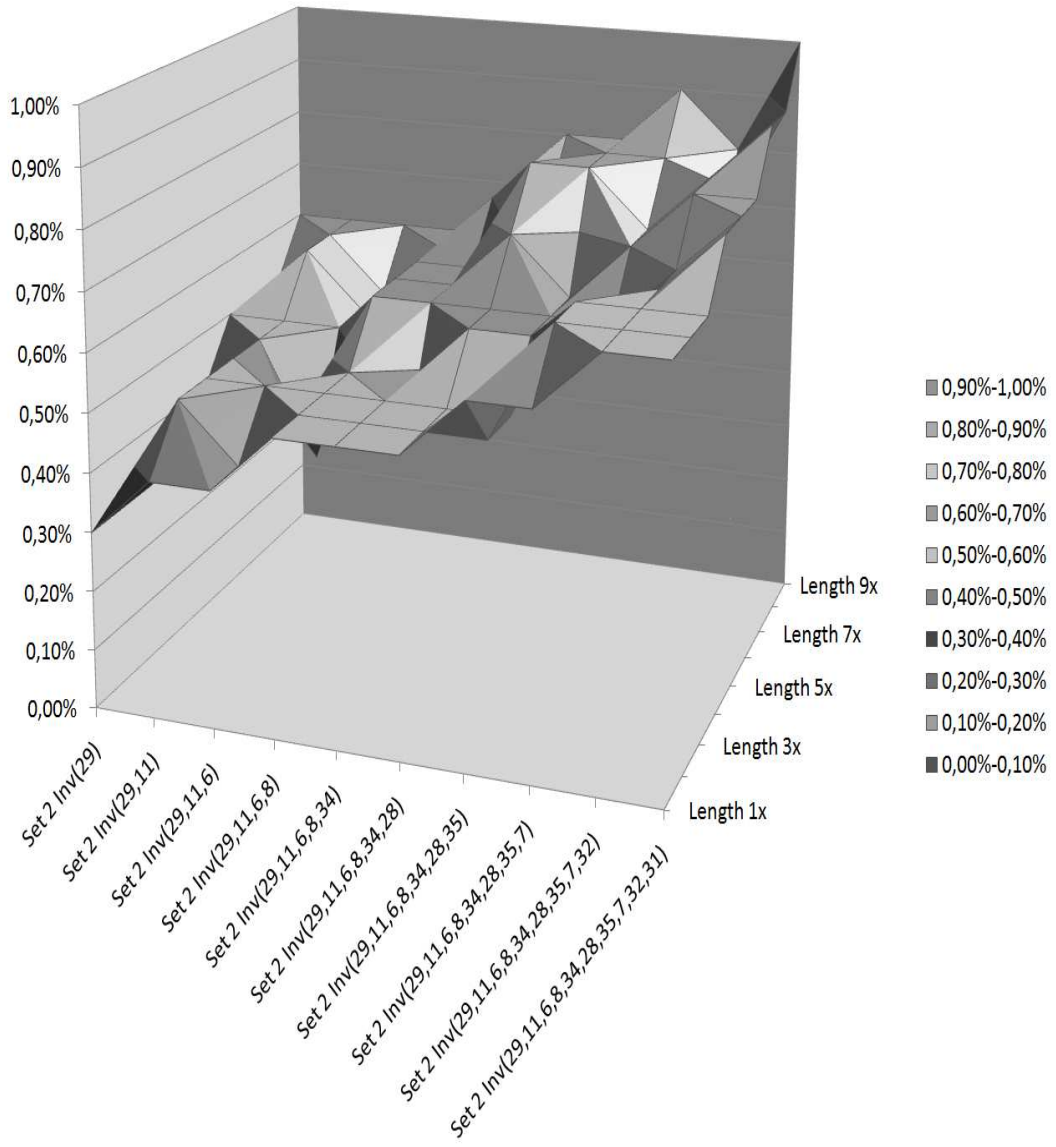


Figure 5.12: Relation between invariants, length of the log, and proportion of errors detected in a tree-like specification.

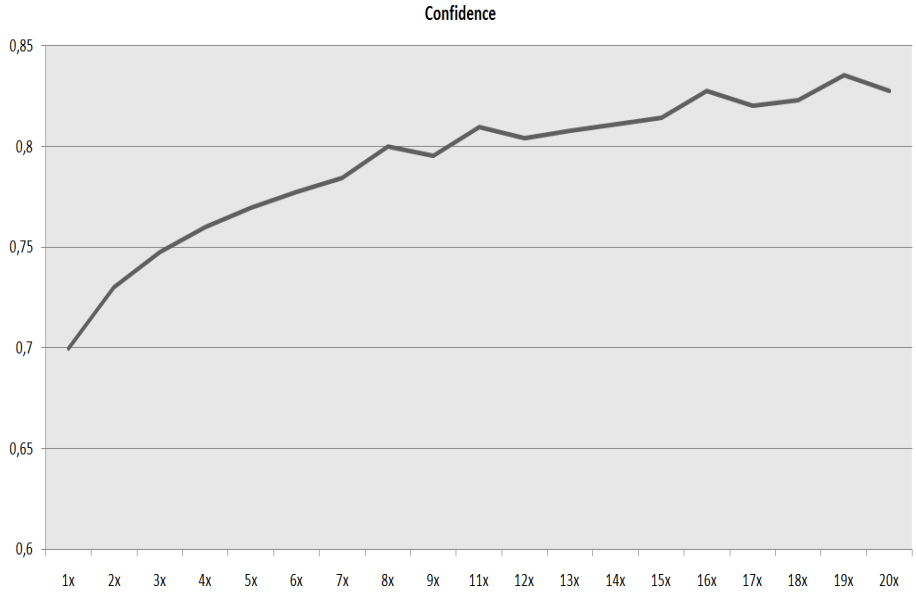


Figure 5.13: Values of confidence using stochastic time passive testing approach.

Chapter 6

Conclusions and Future Work

In this chapter we review the more important aspects of the work presented in Chapters 4 and 5. We also sketch the main research studies we plan to undertake in the future. Shown the significant role that formal methods can play in the timed testing area. We have seen how different formal representations of systems allow us to test their correctness with respect to a specification. Depending on the characterization of the systems we are treating, particular approaches can be more adequate.

In this work we have concentrated on two quantitative extensions to perform formal testing of timed systems. These approaches are based in the idea of *invariants*. The main contribution of our novel frameworks is the integration of different time domains in a single formalism. Specifically, we use a uniform formalism to describe systems where time requirements can be expressed either by using fix time values or by using stochastic time values. This is the purpose of Chapter 4. These formalisms use an extension of *Finite State Machines*. We use $\texttt{TFSM}_{ft}$ and $\texttt{TFSM}_{st}$, instead of FSMs, because we consider the time that an event needs to be performed. We have defined algorithms to detect the correctness of an invariant with respect to a specification and algorithms to detect the correctness with respect to invariants and traces. These algorithms take into account the special features of each time domain that we have studied. While the first setting, using fix time values, is relatively standard, if time conditions are expressed like in the second setting, by means of stochastic functions, we need to apply a method based on a set of observations obtained from the interaction with the implementation. We applied a hypothesis contrast for fixing the similarity level of the random variable extracted from the specification and the observed time values.

In addition to the theoretical framework we have developed a tool called PasTe that

helps in the automation of our passive testing approaches. In particular, all algorithms presented in this Master Thesis are fully implemented, and we present some interesting experiments.

As future theoretical work we plan to improve the capability of our frameworks by adding new classes of invariants. We also would like to increase the classification of specifications that we have mentioned while presenting PasTe. In addition, by using real implementations to test, we plan to formally study a real security protocol by keeping an adequate set of invariants. Another research line is to adequate some other approaches investigated during this year to increasing the power of error detection (applying it to a real time spam filter [AN08] and to a theoretical user-implementer model framework [ALR08]).

Bibliography

[AAD79] J.M. Ayache, P. Azema, and M. Diaz. Observer: A concept for on-line detection of control errors in concurrent systems. In *9th Symposium on Fault-Tolerant Computing*, 1979.

[ABC+95] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, 1995.

[ACB84] M. Ajmone Marsan, G. Conte, and G. Balbo. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, 5(2):93–122, 1984.

[ACC+04] B. Alcalde, A.R. Cavalli, D. Chen, D. Khuu, and D. Lee. Network protocol system passive testing for fault management: A backward checking approach. In *FORTE 2004, LNCS 3235*, pages 150–166. Springer, 2004.

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. Model checking in dense real time. *Information and Computation*, 104:2–34, 1993.

[ACH94] R. Alur, C. Courcoubetis, and T.A. Henzinger. The observational power of clocks. In *5th Int. Conf. on Concurrency Theory, CONCUR'94, LNCS 836*, pages 162–177. Springer, 1994.

[ACN03] J.A. Arnedo, A. Cavalli, and M. Núñez. Fast testing of critical properties through passive testing. In *15th Int. Conf. on Testing Communicating Systems, TestCom'03, LNCS 2644*, pages 295–310. Springer, 2003.

[AD90] R. Alur and D. Dill. Automata for modeling real-time systems. In *17th Int. Colloquium on Automata, Languages and Programming, ICALP'90, LNCS 443*, pages 322–335. Springer, 1990.

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AFH94] R. Alur, L. Fix, and T. Henzinger. A determinizable class of timed automata. In *6th Int. Conf. on Computer Aided Verification, CAV'94, LNCS 818*, pages 1–13. Springer, 1994.

[AH94] R. Alur and T.A. Henzinger. Real-time system = discrete time + clock variables. In *Theories and Experiences for Real-Time System Development, 1st AMAST Workshop on Real-Time System Development*, pages 1–29. World Scientific, 1994.

[ALR08] C. Andrés, L. Llana, and I. Rodríguez. Formally comparing user and implementer model-based testing methods. In *4th Workshop on Advances in Model Based Testing, A-MOST'08*, pages 1–10. IEEE Computer Society Press, 2008.

[AMN08] C. Andrés, M.G. Merayo, and M. Núñez. Passive testing of timed systems. In *6th Int. Symposium on Automated Technology for Verification and Analysis, ATVA'08 (in press)*. Springer, 2008.

[AN08] C. Andrés and M. Núñez. ACABARASE: An Anti-spam CAse-BAsed ReAsoning SystEm. In *3rd Int. Conf. on Systems, ICONS'08*, pages 230–234. IEEE Computer Society Press, 2008.

[AP94] A. Aamodt and E. Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *Artificial Intelligence Communications*, 47(1):39–52, 1994.

[BB91] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.

[BB96] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra. *Formal Aspects of Computing*, 8(2):188–208, 1996.

[BB04] L. Brandán Briones and E. Brinksma. A test generation framework for quiescent real-time systems. In *4th Int. Workshop on Formal Approaches to Testing of Software, FATES'04, LNCS 3395*, pages 64–78. Springer, 2004.

[BB05] L. Brandán Briones and E. Brinksma. Testing real-time multi input-output systems. In *7th Int. Conf. on Formal Engineering Methods, ICFEM'05, LNCS 3785*, pages 264–279. Springer, 2005.

[BBC+02] J.P. Bowen, K. Bogdanov, J. Clark, M. Harman, R.M. Hierons, and P. Krause. Fortest: Formal methods and testing. In *26th IEEE Computer Software and Applications, COMPSAC'02*, pages 91–101. IEEE Computer Society Press, 2002.

[BBG98] M. Bravetti, M. Bernardo, and R. Gorrieri. Towards performance evaluation with general distributions in process algebras. In *9th Int. Conf. on Concurrency Theory, CONCUR'98, LNCS 1466*, pages 405–422. Springer, 1998.

[BBS95] J.C.M. Baeten, J.A. Bergstra, and S.A. Smolka. Axiomatizing probabilistic processes: ACP with generative probabilities. *Information and Computation*, 121(2):234–255, 1995.

[BC00] M. Bernardo and W.R. Cleaveland. A theory of testing for markovian processes. In *11th Int. Conf. on Concurrency Theory, CONCUR'2000, LNCS 1877*, pages 305–319. Springer, 2000.

[BCNZ05] E. Bayse, A. Cavalli, M. Núñez, and F. Zaïdi. A passive testing approach based on invariants: Application to the WAP. *Computer Networks*, 48(2):247–266, 2005.

[BCSS98] M. Bernardo, W.R. Cleaveland, S.I. Sims, and W.J. Stewart. TwoTowers: A tool integrating functional and performance analysis of concurrent systems. In *Formal Description Techniques for Distributed Systems and Communication Protocols (XI), and Protocol Specification, Testing, and Verification (XVIII)*, pages 457–467. Kluwer Academic Publishers, 1998.

[BD04] M. Bravetti and P.R. D'Argenio. Tutte le algebre insieme: Concepts, discussions and relations of stochastic process algebras with general distributions. In *Validation of Stochastic Systems, LNCS 2925*, pages 44–88. Springer, 2004.

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. KRONOS: A model-checking tool for real-time systems. In *10th Int. Conf. on Computer Aided Verification, CAV'98, LNCS 1427*, pages 546–550. Springer, 1998.

[BG98] M. Bernardo and R. Gorrieri. A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theoretical Computer Science*, 202(1-2):1–54, 1998.

[BG02] M. Bravetti and R. Gorrieri. The theory of interactive generalized semi-Markov processes. *Theoretical Computer Science*, 282(1):5–32, 2002.

[BGK+96] J. Bengtsson, D. Griffioen, K. Kristoffersen, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using UPPAAL. In *8th Int. Conf. on Computer Aided Verification, CAV'96*, pages 244–256, 1996.

[BHKR95] S. Bradley, W. Henderson, D. Kendall, and A. Robson. Verification, validation and implementation of timed protocols using AORTA. In *15th WG6.1 Int. Conf. on Protocol Specification, Testing, and Verification, PSTV'95*, pages 205–220. Chapman & Hall, 1995.

[BHS89] A. Bouloutas, G. Hart, and M. Schwartz. On the design of observers for failure detection of discrete event systems. In *Network Management and Control Workshop*. IEEE Computer Society Press, 1989.

[BKLL95] E. Brinksma, J.-P. Katoen, R. Langerak, and D. Latella. A stochastic causality-based process algebra. *The Computer Journal*, 38(7):553–565, 1995.

[BL97] P. Brémond-Grégoire and I. Lee. A process algebra of communicating shared resources with dense time and priorities. *Theoretical Computer Science*, 189(1-2):179–219, 1997.

[BLG93] P. Brémond-Grégoire, I. Lee, and R. Gerber. ACSR: An algebra of communicating shared resources with dense time and priorities. In *4th Int. Conf. on Concurrency Theory, CONCUR'93, LNCS 715*, pages 417–431. Springer, 1993.

[BLL+96] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and W.Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems III, LNCS 1066*, pages 232–243. Springer, 1996.

[BLL+98] J. Bengtsson, K.G. Larsen, F. Larsson, P. Petterson, W. Yi, and C. Weise. New generation of UPPAAL. In *Int. Workshop on Software Tools and Technology Transfer*, 1998.

[BM77] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20:762–772, 1977.

[BU91a] B.S. Bosik and M.Ü. Uyar. Finite state machine based formal methods in protocol conformance testing. *Computer Networks & ISDN Systems*, 22:7–33, 1991.

[BU91b] S.C. Boyd and H. Ural. On the complexity of generating optimal test sequences. *IEEE Transactions on Software Engineering*, 17(9):976–978, 1991.

[Buc94] P. Buchholz. Markovian process algebra: Composition and equivalence. In *2nd Workshop on Process Algebra and Performance Modelling, PAPM'94*, pages 11–30, 1994.

[BVC05] A. Bueno, V. Valero, and F. Cuartero. A translation of TPAL_p into a class of timed-probabilistic Petri nets. *Theoretical Computer Science*, 338(1–3):350–392, 2005.

[Car99] R. Cardell-Oliver. Conformance testing of real-time systems against timed automata specifications. Technical Report CSM-330, University of Essex, 1999.

[Car00] R. Cardell-Oliver. Conformance tests for real-time systems with timed automata specifications. *Formal Aspects of Computing*, 12(5):350–371, 2000.

[Cas93] C.G. Cassandras. *Discrete Event Systems. Modeling and Performance Analysis.* Aksen Associates - Irwin, 1993.

[CCV+03] D. Cazorla, F. Cuartero, V. Valero, F.L. Pelayo, and J.J. Pardo. Algebraic theory of probabilistic and non-deterministic processes. *Journal of Logic and Algebraic Programming*, 55(1–2):57–103, 2003.

[CCVP01] D. Cazorla, F. Cuartero, V. Valero, and F.L. Pelayo. A process algebra for probabilistic and nondeterministic processes. *Information Processing Letters*, 80:15–23, 2001.

[CG98] R. Cardell-Oliver and T. Glover. A practical and complete algorithm for testing real-time systems. In *5th Int. Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT'98, LNCS 1486*, pages 251–260. Springer, 1998.

[CGP01] A. Cavalli, C. Gervy, and S. Prokopenko. New approaches for passive testing using an extended finite state machine specification. In *Concordia Prestigious Workshop on Communication Software Engineering*, pages 225–250, 2001.

[CGP03] A. Cavalli, C. Gervy, and S. Prokopenko. New approaches for passive testing using an extended finite state machine specification. *Journal of Information and Software Technology*, 45:837–852, 2003.

[Cho78] T.S. Chow. Testing software design modelled by finite state machines. *IEEE Transactions on Software Engineering*, 4:178–187, 1978.

[CHR91] Z. Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40:269–276, 1991.

[CKL97] D. Clarke, Y.S. Kim, and I. Lee. Automatic test generation for the analysis of a real-time system: Case study. In *3rd IEEE Real Time Technology and Applications Symposium, RTAS'97*, pages 112–124. IEEE Computer Society Press, 1997.

[CKL98] R. Castanet, O. Koné, and P. Laurençot. On the fly test generation for real-time protocols. In *7th Int. Conf. on Computer Communications and Networks, IC3N'98*, pages 378–387. IEEE Computer Society Press, 1998.

[CKXI03] S. Chen, Z. Kalbarczyk, J. Xu, and R. K. Iyer. A data-driven finite state machine model for analyzing security vulnerabilities. In *33rd Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks, DSN'03*, pages 605–614, 2003.

[CL97] D. Clarke and I. Lee. Automatic generation of tests for timing constraints from requirements. In *3rd Workshop on Object-Oriented Real-Time Dependable Systems, WORDS'97*, pages 199–206. IEEE Computer Society Press, 1997.

[CV06] A. Cavalli and D. Vieira. An enhanced passive testing approach for network protocols. In *Int. Conf. on Networking, Int. Conf. on Systems, and Int. Conf. on Mobile Communications and Learning Technologies, ICN/ICONS/ICMCL'06*, page 169. IEEE Computer Society Press, 2006.

[Dij72] E.W. Dijkstra. Notes of structured programming. In O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, *Structured Programming*. Academic Press, 1972.

[DK05] P.R. D'Argenio and J.-P. Katoen. A theory of stochastic systems part II: Process algebra. *Information and Computation*, 203(1):39–74, 2005.

[DKRT97] P.R. D'Argenio, J.-P. Katoen, T.C. Ruys, and J. Tretmans. The bounded retransmission protocol must be on time! In *3rd Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'97, LNCS 1217*, pages 416–431. Springer, 1997.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool Kronos. In *Hybrid Systems III, LNCS 1066*, pages 208–219. Springer, 1996.

[DOY95] C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with KRONOS. In *7th IFIP WG6.1 Int. Conf. on Formal Description Techniques, FORTE'94*, pages 227–242. Chapman & Hall, 1995.

[DU04] A.Y. Duale and M.Ü. Uyar. A method enabling feasible conformance test sequence generation for EFSM models. *IEEE Transactions on Computers*, 53(5):614–627, 2004.

[DY81] D. Dolev and A. C. Yao. On the security of public key protocols. Technical Report CS-TR-81-854, Stanford University, 1981.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *16th IEEE Real-Time Systems Symposium, RTSS'95*, pages 66–75. IEEE Computer Society Press, 1995.

[ED03] A. En-Nouaary and R. Dssouli. A guided method for testing timed input output automata. In *15th Int. Conf. on Testing Communicating Systems, TestCom'03, LNCS 2644*, pages 211–225. Springer, 2003.

[EDK02] A. En-Nouaary, R. Dssouli, and F. Khendek. Timed Wp-method: Testing real time systems. *IEEE Transactions on Software Engineering*, 28(11):1024–1039, 2002.

[EDKE98] A. En-Nouaary, R. Dssouli, F. Khendek, and A. Elqortobi. Timed test cases generation based on state characterization technique. In *19th IEEE Real Time Systems Symposium, RTSS'98*, pages 220–231. IEEE Computer Society Press, 1998.

[ES00] N. Evans and S. Schneider. Analysing time dependent security properties in CSP using PVS. In *6th European Symposium on Research in Computer Security, ESORICS'00, LNCS 1895*, pages 222–237. Springer, 2000.

[FBK+91] S. Fujiwara, G. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite-state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.

[FNQ95] D. de Frutos, M. Núñez, and J. Quemada. Characterizing termination in LOTOS via testing. In *15th WG6.1 Int. Conf. on Protocol Specification, Testing, and Verification, PSTV'95*, pages 237–250. Chapman & Hall, 1995.

[FPS01] H. Fouchal, E. Petitjean, and S. Salva. An user-oriented testing of real time systems. In *IEEE Workshop on Real-Time Embedded Systems, RTES'01*. IEEE Computer Society Press, 2001.

[Gau95] M.-C. Gaudel. Testing can be formal, too! In *6th Int. Joint Conf. CAAP/FASE, Theory and Practice of Software Development, TAPSOFT'95, LNCS 915*, pages 82–96. Springer, 1995.

[GH02] D. Geer and J. Harthorne. Penetration testing: A duet. In *18th Annual Computer Security Applications Conference, ACSAC'02*, pages 185–198. IEEE Computer Society Press, 2002.

[GHR93] N. Götz, U. Herzog, and M. Rettelbach. Multiprocessor and distributed system design: The integration of functional specification and performance analysis using stochastic process algebras. In *16th Int. Symp. on Computer Performance Modelling, Measurement and Evaluation, PERFORMANCE'93, LNCS 729*, pages 121–146. Springer, 1993.

[GJS90] A. Giacalone, C.-C. Jou, and S.A. Smolka. Algebraic reasoning for probabilistic concurrent systems. In *IFIP WG2.3 Conf. on Programming Concepts and Methods*. North Holland, 1990.

[Gly89] P.W. Glynn. A GSMP formalism for discrete event simulation. *Proceedings of the IEEE*, 77(1):14–23, 1989.

[Gro91] J.F. Groote. Specification and verification of real time systems in ACP. In *10th WG6.1 Int. Conf. on Protocol Specification, Testing, and Verification, PSTV'90*, pages 261–274. North-Holland, 1991.

[GSS95] R. van Glabbeek, S.A. Smolka, and B. Steffen. Reactive, generative and stratified models of probabilistic processes. *Information and Computation*, 121(1):59–80, 1995.

[Han91] H. Hansson. *Time and Probability in Formal Design of Distributed Systems*. PhD thesis, Department of Computer Systems. Uppsala University, 1991.

[HBB+08] R.M. Hierons, K. Bogdanov, J.P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luettgen, A.J.H Simons, S. Vilkomir, M.R. Woodward, and H. Zedan. Using formal methods to support testing. *ACM Computing Surveys (in press)*, 2008.

[Hen64] F.C. Hennie. Fault-detecting experiments for sequential circuits. In *5th Annual Symposium on Switching Circuit Theory and Logical Design*, pages 95–110, 1964.

[Her90] U. Herzog. Formal description, time and performance analysis. a framework. In T. Härder, H. Wedekind, and G. Zimmermann, editors, *Entwurf und Betrieb verteilter Systeme, Fachtagung des Sonderforschungsbereiche 124 und 182*, pages 172–190. Springer, 1990.

[Her98] H. Hermanns. *Interactive Markov Chains.* PhD thesis, Universität Erlangen-Nürnberg, 1998. Also appeared as *LNCS 2428*, Springer, 2002.

[HHWT95] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: The next generation. In *16th IEEE Real-Time Systems Symposium, RTSS'95*, pages 55–65. IEEE Computer Society Press, 1995.

[HI98] M. Holcombe and F. Ipate. *Correct Systems: Building a Business Process Solution.* Springer, 1998.

[Hil94] J. Hillston. The nature of synchronization. In *2nd Workshop on Process Algebra and Performance Modeling, PAPM'94*, pages 51–70. University of Erlangen, 1994.

[Hil96] J. Hillston. *A Compositional Approach to Performance Modelling.* Cambridge University Press, 1996.

[HLN+03] A. Hessel, K. Larsen, B. Nielsen, P. Petterson, and A. Skou. Time-optimal real-time test case generation using UPPAAL. In *3rd Int. Workshop on Formal Approaches to Testing of Software, FATES'03, LNCS 2931*, pages 114–130. Springer, 2003.

[HLSV00] R. Hao, D. Lee, R. K. Sinha, and D. Vlah. Testing IP routing protocols - from probabilistic algorithms to a software tool. In *20th Joint Int. Conf. on Protocol Specification, Testing, and Verification and Formal Description Techniques, FORTE/PSTV'00*, pages 249–264. Kluwer Academic Publishers, 2000.

[HMP92] Y. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *19th International Colloquium on Automata, Languages, and Programming, ICALP'92, LNCS 632*, pages 545–558, 1992.

[HNSY94] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:193–244, 1994.

[HNTC99] T. Higashino, A. Nakata, K. Taniguchi, and A. Cavalli. Generating test cases for a timed I/O automaton model. In *12th Int. Workshop on Testing of Communicating Systems, IWTCS'99*, pages 197–214. Kluwer Academic Publishers, 1999.

[Hoa96] C.A.R. Hoare. How did software get so reliable without proof? In *3rd. Int. Symposium of Formal Methods Europe, FME'96, LNCS 1051*, pages 1–17. Springer, 1996.

[HP92] P.G. Harrison and N.M. Patel. *Performance Modelling of Communication Networks and Computer Architectures.* Addison-Wesley, 1992.

[HR94] H. Hermanns and M. Rettelbach. Syntax, semantics, equivalences, and axioms for MTIPP. In *2nd Workshop on Process Algebra and Performance Modelling*, pages 71–88, 1994.

[HS95] P.G. Harrison and B. Strulo. Stochastic process algebra for discrete event simulation. In *Quantitative Methods in Parallel Systems*, pages 18–37. Springer, 1995.

[HW05] G.-D. Huang and F. Wang. Automatic test case generation with region-related coverage annotations for real-time systems. In *3rd Int. Symposium on Automated Technology for Verification and Analysis, ATVA'05, LNCS 3707*, pages 144–158. Springer, 2005.

[HW08] P. E. Haxell and G. T. Wilfong. A fractional model of the border gateway protocol (BGP). In *19th Annual ACM-SIAM Symposium on Discrete algorithms, SODA'08*, pages 193–199. Society for Industrial and Applied Mathematics, 2008.

[HWT95] P.-H. Ho and H. Wong-Toi. Automated analysis of an audio control protocol. In *7th Int. Conf. on Computer Aided Verification, CAV'95, LNCS 939*, pages 381–394. Springer, 1995.

[IH97] F. Ipate and M. Holcombe. An integration testing method that is proved to find all faults. *International Journal of Computer Mathematics*, 63(3-4):159–178, 1997.

[IT97] ITU-T. *Recommendation Z.500 Framework on formal methods in conformance testing*. International Telecommunications Union, Geneva, Switzerland, 1997.

[Kho02] A. Khoumsi. A method for testing the conformance of real-time systems. In *7th Int. Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT'02, LNCS 2469*, pages 331–354. Springer, 2002.

[KJM03] A. Khoumsi, T. Jéron, and H. Marchand. Test cases generation for nondeterministic real-time systems. In *3rd Int. Workshop on Formal Approaches to Testing of Software, FATES'03, LNCS 2931*, pages 131–146. Springer, 2003.

[Kle75] L. Kleinrock. *Queueing Systems*. John Wiley & Sons, 1975.

[KMP77] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.

[KT04] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In *11th Int. SPIN Workshop on Model Checking of Software, SPIN'04, LNCS 2989*, pages 109–126. Springer, 2004.

[KT05] M. Krichen and S. Tripakis. An expressive and implementable formal framework for testing real-time systems. In *17th Int. Conf. on Testing of Communicating Systems, TestCom'05, LNCS 3502*, pages 209–225. Springer, 2005.

[Lai02] R. Lai. A survey of communication protocol testing. *Journal of Systems and Software*, 62:21–46, 2002.

[Lan90] R. Langerak. A testing theory for LOTOS using deadlock detection. In *9th WG6.1 Int. Conf. on Protocol Specification, Testing, and Verification, PSTV'89*, pages 87–98. North Holland, 1990.

[LCH+02] D. Lee, D. Chen, R. Hao, R. Miller, J. Wu, and X. Yin. A formal approach for passive testing of protocol data portions. In *10th IEEE Int. Conf. on Network Protocols, ICNP'02*, pages 122–131. IEEE Computer Society Press, 2002.

[LCH+06] D. Lee, D. Chen, R. Hao, R.E. Miller, J. Wu, and X. Yin. Network protocol system monitoring: a formal approach with passive testing. *IEEE/ACM Transactions on Networking*, 14:424–437, 2006.

[LL97] L. Léonard and G. Leduc. An introduction to ET-LOTOS for the description of time-sensitive systems. *Computer Networks and ISDN Systems*, 29:271–292, 1997.

[LN00] N. López and M. Núñez. `NMSPA`: A non-markovian model for stochastic processes. In *International Workshop on Distributed System Validation and Verification (DSVV'2000)*, pages 33–40, 2000.

[LN01] N. López and M. Núñez. A testing theory for generally distributed stochastic processes. In *12th Int. Conf. on Concurrency Theory, CONCUR'01, LNCS 2154*, pages 321–335. Springer, 2001.

[LNR04] N. López, M. Núñez, and F. Rubio. An integrated framework for the analysis of asynchronous communicating stochastic processes. *Formal Aspects of Computing*, 16(3):238–262, 2004.

[LNS+97] D. Lee, A.N. Netravali, K.K. Sabnani, B. Sugla, and A. John. Passive testing and applications to network management. In *5th IEEE Int. Conf. on Network Protocols, ICNP'97*, pages 113–122. IEEE Computer Society Press, 1997.

[Low95] G. Lowe. Probabilistic and prioritized models of timed CSP. *Theoretical Computer Science*, 138:315–352, 1995.

[Low96] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *2nd Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems, TACAS'96, LNCS 1055*, pages 147–166. Springer, 1996.

[LPY97] K.G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[LPY98] M. Lindahl, P. Pettersson, and W. Yi. Formal design and analysis of a gear controller. In *4th Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'98, LNCS 1384*, pages 281–297. Springer, 1998.

[LS94] S. Lee and K. G. Shin. Probabilistic diagnosis of multiprocessor systems. *ACM Computer Surveys*, 26(1):121–139, 1994.

[LSK+93] D. Lee, K.K. Sabnani, D.M. Kristol, S. Paul, and M.Ü. Uyar. Conformance testing of protocols specified as communicating FSMs. In *12th Annual Joint Conf. of the IEEE Computer and Communications Societies. Networking: Foundation for the Future, INFOCOM'93*, pages 115–127. IEEE Computer Society Press, 1993.

[LSKP96] D. Lee, K.K. Sabnani, D.M. Kristol, and S. Paul. Conformance testing of protocols specified as communicating finite state machines - a guided random walk based approach. *IEEE Transactions on Communications*, 44(5):631–640, 1996.

[LSV03] N.A. Lynch, R. Segala, and F.W. Vaandrager. Hybrid I/O automata. *Information and Computation*, 185(1):105–157, 2003.

[LV96] N.A. Lynch and F.W. Vaandrager. Action transducers and timed automata. *Formal Aspects of Computing*, 8(5):499–538, 1996.

[LY96] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines: A survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[MA00] R.E. Miller and K.A. Arisha. On fault location in networks by passive testing. In *19th IEEE Int. Performance, Computing, and Communications Conference, IPCCC'00*, pages 281–287. IEEE Computer Society Press, 2000.

[MA01] R.E. Miller and K.A. Arisha. Fault identification in networks by passive testing. In *34th Simulation Symposium, SS'01*, pages 277–284. IEEE Computer Society Press, 2001.

[Mea92] C. Meadows. Applying formal methods to the analysis of a key management protocol. *Journal of Computer Security*, 1(1):5–36, 1992.

[Mea03] C. Meadows. Formal methods for cryptographic protocol analysis: emerging issues and trends. *IEEE Journal on Selected Areas in Communications*, 21(1):44–54, 2003.

[Mil98] R.E. Miller. Passive testing of networks using a CFSM specification. In *IEEE Int. Performance Computing and Communications Conference*, pages 111–116. IEEE Computer Society Press, 1998.

[MMM95] D. Mandrioli, S. Morasca, and A. Morzenti. Generating test cases for real time systems from logic specifications. *ACM Transactions on Computer Systems*, 13(4):356–398, 1995.

[MNR06] M.G. Merayo, M. Núñez, and I. Rodríguez. Extending EFSMs to specify and test timed systems with action durations and timeouts. In *26th IFIP WG 6.1 Int. Conf. on Formal Techniques for Networked and Distributed Systems, FORTE'06, LNCS 4229*, pages 372–387. Springer, 2006.

[MNR07a] M.G. Merayo, M. Núñez, and I. Rodríguez. A brief introduction to $\mathcal{THOTL}$. In *5th Int. Symposium on Automated Technology for Verification and Analysis, ATVA'07, LNCS 4762*, pages 501–510. Springer, 2007.

[MNR07b] M.G. Merayo, M. Núñez, and I. Rodríguez. Formal testing of systems presenting soft and hard deadlines. In *2nd IPM Int. Symposium on Fundamentals of Software Engineering, FSEN'07, LNCS 4767*, pages 160–174. Springer, 2007.

[MNR08a] M.G. Merayo, M. Núñez, and I. Rodríguez. $\mathcal{THOTL}$: A timed extension of $\mathcal{HOTL}$. In *Joint 20th IFIP TC6/WG6.1 Int. Conf. on Testing of Software and Communicating Systems, TestCom'08, and 8th Int. Workshop on Formal Approaches to Software Testing, FATES'08, LNCS 5047*, pages 86–102. Springer, 2008.

[MNR08b] M.G. Merayo, M. Núñez, and I. Rodríguez. Extending EFSMs to specify and test timed systems with action durations and timeouts. *IEEE Transactions on Computers*, 57(6):835–848, 2008.

[MNR08c] M.G. Merayo, M. Núñez, and I. Rodríguez. Formal testing from timed finite state machines. *Computer Networks*, 52(2):432–460, 2008.

[Moo56] E.P. Moore. Gedanken experiments on sequential machines. In C. Shannon and J. McCarthy, editors, *Automata Studies*. Princeton University Press, 1956.

[MP93] R.E. Miller and S. Paul. On the generation of minimal-length conformance tests for communication protocols. *IEEE/ACM Transactions on Networking*, 1(1):116–129, 1993.

[MP94] R.E. Miller and S. Paul. Structural analysis of protocol specifications and generation of maximal fault coverage conformance test sequences. *IEEE/ACM Transactions on Networking*, 2(5):457–470, 1994.

[MT90] F. Moller and C. Tofts. A temporal calculus of communicating systems. In *1st Int. Conf. on Concurrency Theory, CONCUR'90, LNCS 458*, pages 401–415. Springer, 1990.

[MY96] O. Maler and S. Yovine. Hardware timing verification using KRONOS. In *7th IEEE Israeli Conf. on Computer Systems and Software Engineering, ICCB-SSE'96*, pages 23–29. IEEE Computer Society Press, 1996.

[NF95] M. Núñez and D. de Frutos. Testing semantics for probabilistic LOTOS. In *8th IFIP WG6.1 Int. Conf. on Formal Description Techniques, FORTE'95*, pages 365–380. Chapman & Hall, 1995.

[NFL95] M. Núñez, D. de Frutos, and L. Llana. Acceptance trees for probabilistic processes. In *6th Int. Conf. on Concurrency Theory, CONCUR'95, LNCS 962*, pages 249–263. Springer, 1995.

[NR99] M. Núñez and D. Rupérez. Fair testing through probabilistic testing. In *Formal Description Techniques for Distributed Systems and Communication Protocols (XII), and Protocol Specification, Testing, and Verification (XIX)*, pages 135–150. Kluwer Academic Publishers, 1999.

[NR03] M. Núñez and I. Rodríguez. Towards testing stochastic timed systems. In *23rd IFIP WG 6.1 Int. Conf. on Formal Techniques for Networked and Distributed Systems, FORTE'03, LNCS 2767*, pages 335–350. Springer, 2003.

[NR06] M. Núñez and I. Rodríguez. Conformance testing relations for timed systems. In *5th Int. Workshop on Formal Approaches to Software Testing, FATES'05, LNCS 3997*, pages 103–117. Springer, 2006.

[NS78] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[NS91] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *3rd Int. Conf. on Computer Aided Verification, CAV'91, LNCS 575*, pages 376–398. Springer, 1991.

[NS01] B. Nielsen and A. Skou. Automated test generation from timed automata. In *7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'01, LNCS 2031*, pages 343–357. Springer, 2001.

[NSY92] X. Nicollin, J. Sifakis, and S. Yovine. Compiling real-time specifications into extended automata. *IEEE Transactions on Software Engineering*, 18(9):794–804, 1992.

[Núñ03] M. Núñez. Algebraic theory of probabilistic processes. *Journal of Logic and Algebraic Programming*, 56(1–2):117–177, 2003.

[NVN98] G. Noubir, K. Vijayananda, and H. J. Nussbaumer. Signature-based method for run-time fault detection in communication protocols. *Computer Communications*, 21(5):405–421, 1998.

[PCVM04] F.L. Pelayo, F. Cuartero, V. Valero, and H. Macià. Applying timed-arc petri nets to improve the performance of the MPEG–2 encoding algorithm. In *10th Int. Conf. on Multimedia Modelling (MMM'04)*, pages 49–56. IEEE Computer Society Press, 2004.

[PF99] E. Petitjean and H. Fouchal. From timed automata to testable untimed automata. In *24th IFAC/IFIP International Workshop on Real-Time Programming, WRTP'99*. Elsevier, 1999.

[Pri96] C. Priami. Stochastic π-calculus with general distributions. In *4th Int. Workshop on Process Algebra and Performance Modelling, PAPM'96*, 1996.

[PS97] J. Peleska and M. Siegel. Test automation of safety-critical reactive systems. *South African Computer Journal*, 19:53–77, 1997.

[QFA93] J. Quemada, D. de Frutos, and A. Azcorra. TIC: A TImed Calculus. *Formal Aspects of Computing*, 5:224–252, 1993.

[RMN08] I. Rodríguez, M.G. Merayo, and M. Núñez. $\mathcal{HOTL}$: Hypotheses and observations testing logic. *Journal of Logic and Algebraic Programming*, 74(2):57–93, 2008.

[RNHW98] P. Raymond, X. Nicollin, N. Halbwatchs, and D. Waber. Automatic testing of reactive systems. In *19th IEEE Real Time Systems Symposium, RTSS'98*, pages 200–209. IEEE Computer Society Press, 1998.

[Ros81] E.C. Rosen. Vulnerabilities of network control protocols: an example. *SIGSOFT Software Engineering Notes*, 6(1):6–8, 1981.

[SD88] K. Sabnani and A. Dahbura. A protocol test generation procedure. *Computer Networks and ISDN Systems*, 15:285–297, 1988.

[SDJ+92] S. Schneider, J. Davies, D. M. Jackson, G.M. Reed, J.N. Reed, and A.W. Roscoe. Timed CSP: Theory and practice. In *Real-Time: Theory in Practice, REX Workshop, LNCS 600*, pages 640–675. Springer, 1992.

[Sei72] C.L. Seitz. An approach to designing checking experiments based on a dynamic model. *Theory of Machines and Computations*, 6(1):341–9, 1972.

[SGSL98] R. Segala, R. Gawlick, J.F. Søgaard-Andersen, and N. Lynch. Liveness in timed and untimed systems. *Information and Computation*, 141:119–171, 1998.

[She93] G.S. Shedler. *Regenerative Stochastic Simulation*. Academic Press, 1993.

[SHJ+02] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *23rd IEEE Symposium on Security and Privacy, SP'02*, pages 273–284. IEEE Computer Society, 2002.

[Sif77] J. Sifakis. Use of Petri nets for performance evaluation. In *3rd Int. Symposium on Measuring, Modelling and Evaluating Computer Systems*, pages 75–93. North-Holland, 1977.

[SL95] D. Sidhu and T. Leung. Fault coverage of protocol test methods. In R.J. Linn and M.Ü. Uyar, editors, *Conformance testing methodologies and architectures for OSI protocols*, pages 449–454. IEEE Computer Society Press, 1995.

[SL06] G. Shu and D. Lee. Message confidentiality testing of security protocols - passive monitoring and active checking. In *18th Int. Conf. on Testing Communicating Systems, TestCom'06, LNCS 3964*, pages 357–372. Springer, 2006.

[SVD01] J. Springintveld, F. Vaandrager, and P.R. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254(1-2):225–257, 2001. Previously appeared as Technical Report CTIT-97-17, University of Twente, 1997.

[TC99] M. Tabourier and A. Cavalli. Passive testing and application to the GSM-MAP protocol. *Journal of Information and Software Technology*, 41:813–821, 1999.

[TCI99] M. Tabourier, A. Cavalli, and M. Ionescu. A GSM-MAP protocol experiment using passive testing. In *World Congress on Formal Methods in the Development of Computing Systems, FM'99, LNCS 1708*, pages 915–934. Springer, 1999.

[Tho03] H. H. Thompson. Why security testing is hard. *IEEE Security & Privacy*, 1(4):83–86, 2003.

[Tho05] H. H. Thompson. Application penetration testing. *IEEE Security & Privacy*, 3(1):66–69, 2005.

[Tow88] D.M. Tow. Network management - recent advances and future trends. *IEEE Journal on Selected Areas in Communications*, 6(4):732–741, 1988.

[Tre96] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software – Concepts and Tools*, 17(3):103–120, 1996.

[Tre99] J. Tretmans. Testing concurrent systems: A formal approach. In *10th Int. Conf. on Concurrency Theory, CONCUR'99, LNCS 1664*, pages 46–65. Springer, 1999.

[Tre08] J. Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing, LNCS 4949*, pages 1–38. Springer, 2008.

[UXZ07] H. Ural, Z. Xu, and F. Zhang. An improved approach to passive testing of fsm-based systems. In *2nd Int. Workshop on Automation of Software Test, AST'07*, page 6. IEEE Computer Society Press, 2007.

[VCI90] S.T. Voung, W.L. Chan, and M.R. Ito. The UIOv-method for protocol test sequence generation. In *2nd IFIP TC6 Int. Workshop on Protocol Test Systems, IWPTS'89*, pages 161–175. North-Holland, 1990.

[VPC02] V. Valero, J.J. Pardo, and F. Cuartero. Translating TPAL specifications into timed-arc Petri nets. In *Application and Theory of Petri Nets 2002, LNCS 2360*, pages 414–433. Springer, 2002.

[Wes89] C. H. West. Protocol validation in complex systems. *ACM SIGCOMM Computer Communications Review*, 19(4):303–312, 1989.

[Whi80] W. Whitt. Continuity of generalized semi-markov processes. *Mathematics of Operational Research*, 5:494–501, 1980.

[WS93] C. Wang and M. Schwartz. Fault detection with multiple observers. *IEEE/ACM Transactions on Networking*, 1(1):48–55, 1993.

[WSW+07] W. Wei, K. Suh, B. Wang, Y. Gu, J. Kurose, and D. Towsley. Passive online rogue access point detection using sequential hypothesis testing with TCP ACK-pairs. In *7th ACM SIGCOMM Internet Measurement Conference, IMC '07*, pages 365–378. ACM Press, 2007.

[WZY01] J. Wu, Y. Zhao, and X. Yin. From active to passive: Progress in testing of internet routing protocols. In *21st IFIP WG 6.1 Int. Conf. on Formal Techniques for Networked and Distributed Systems, FORTE'01*, pages 101–116. Kluwer Academic Publishers, 2001.

[Yi90] W. Yi. Real-time behavior of asynchronous agents. In *1st Int. Conf. on Concurrency Theory, CONCUR'90, LNCS 458*, pages 502–520. Springer, 1990.

[YL95] M. Yannakakis and D. Lee. Testing finite state machines: fault detection. *Journal of Computer and System Sciences*, 50(2):209–227, 1995.

[YPD95] W. Yi, P. Pettersson, and M. Daniels. Automatic verification of real-time communicating systems by constraint-solving. In *7th IFIP WG6.1 Int. Conf. on Formal Description Techniques, FORTE'94*, pages 243–258. Chapman & Hall, 1995.

[YWS+89] S.W. Yeh, C. Wu, H.D. Sheng, C.K. Hung, and R.C. Lee. Expert system based automatic network fault management system. In *13th Annual Int. Computer Software and Applications Conference, COMPSAC'89*, pages 767–774. IEEE Computer Society Press, 1989.

[Zub80] W.M. Zuberek. Timed Petri nets and preliminary performance evaluation. In *7th Annual Symposium on Computer Architecture*, pages 88–96. ACM Press, 1980.

[ZYW01] Y. Zhao, X. Yin, and J. Wu. OnLine test system, an application of passive testing in routing protocols test. In *9th IEEE Int. Conf. on Networks, ICON'01*, pages 190–195. IEEE Computer Society Press, 2001.

[ZYW03] Y. Zhao, X. Yin, and J. Wu. Problems in the information dissemination of the internet routing. *Journal of Computer Science and Technology*, 18(2):139–152, 2003.