University of Castilla-La Mancha

A publication of the

Department of Computer Science

P_UPPAAL a tool for capturing the probabilistic behaviour of UPPAAL models

by

Gregorio Díaz Descalzo, Fernando Cuartero Gómez and Kim G. Larsen

Technical Report **#DIAB-02-01-33** December 2002

DEPARTAMENTO DE INFORMÁTICA
ESCUELA POLITÉCNICA SUPERIOR
UNIVERSIDAD DE CASTILLA-LA MANCHA
Campus Universitario s/n
Albacete - 02071 - Spain
Phone +34.967.599200, Fax +34.967.599224

ABSTRACT

As real-time systems often operate in safety-critical environments, it is extremely important to know that these systems are correct. A traditional way to guarantee the correctness of a real-time system is by simulation and testing. However, even if the system passes all imaginables tests, it does not guarantee that the system works correctly in all real life situations. This motivates the use of formal methods for modeling systems and verifying that they fulfill their specification. This needs will be discussed in the first chapter.

The second chapter describes networks of timed automata as a formal model for real-time systems; it also describes how to extend this model to handle systems with drifting clocks. A simple logic for expressing safety properties of a system is presented, together with an algorithm for checking whether a model satisfies a given property or not. It is the basis for UPPAAL, a tool for automatic verification. To handle the infinite state-space UPPAAL uses constraint solving techniques to group states having the same properties. UPPAAL also adopt the onthefly verification techniques, to avoid constructing the parts of the state-space that are not reachable.

The third chapter describes a tool named P_UPPAAL. P_UPPAAL allows the system modeling, simulation and verification. The modeling and simulation processes are driven through a graphical interface and the verification process is realized by RAPTURE. This tool, P_UPPAAL, introduces the probabilistic behaviour to UPPAAL models. To add this behaviour, the tool imports models from UPPAAL and translates them into a graphical model. The graphical model is translated to a textual format which is used by RAPTURE engine.

Furthermore, in this section is described RAPTURE, which makes model check of quantitative reachability properties on Markov decision processes together with its prototype implementation. The technique is based on the analysis performed on an abstraction of the model under analysis. Such an abstraction is significantly smaller than the original model and may safely refute or accept

the required property. Otherwise, the abstraction is refined and the process repeated.

The fourth chapter shows the conclusions of this research work and futures researches that we are involved in.

Table of Contents

Chapter 1

INTRODUCTION

This chapter shows the need for system verification. Furthermore, it discusses the formal verification techniques. Therefore, the main objective in this chapter is to introduce the reader in formal methods world and in their advantages.

1.1 The Need for Formal Methods

Today, hardware and software system are widely used in applications where failure is unacceptable: electronic commerce, telephone switching networks, highway and air traffic control systems, medical instruments, and other examples too numerous to list. We frequently read of incidents where some failure is caused by an error in a hardware or software system. A recent example of such a failure is the Ariane 5 rocket, which exploded on June 4, 1996, less than forty seconds after it was launched. The committee that investigated the accident fund that it was caused by a software error in the computer that was responsible for calculating the rocket's movement. During the launch , an exception occurred when a large 64-bit floating point number was converted to a 16-bit signed integer. This conversion was not protected by code for handling exceptions and caused the destruction of the rocket. The team investigating the failure suggested that several measures must be taken in order to prevent similar incidents in the future, including the verification of the Ariane 5 software.

Clearly, the need for reliable hardware an software is critical. As the involvement of such systems in our lives increases, so too does

the burden for ensuring their correctness. Unfortunately, it is no longer feasible to shut down a malfunctioning system in order to restore safety. We are very much dependent on such systems for continuous operation; in fact, in some cases, devices are less safe when they are shut down. Even when failure is not life-threatening, the consequences of having to replace critical code or circuitry can be economically devastating.

Because of the success of the Internet an embedded systems in automobiles, airplanes, and other safety critical systems, we are likely to become even more dependent on the proper functioning of computing devices in the future. In fact, the pace of change will likely accelerate in coming years. Because of this rapid growth in technology, it will become even more important to develop methods than increase our confidence in the correctness of such systems.

1.2 Formal Verifications Techniques

The main problem in the design of many complex systems lies in the effort spent on verification than on construction. Formal methods provide techniques to integrate verification in early stages of system development. In this chapter a survey of this techniques is introduced.

1.2.1 Formal Methods

The formal methods can be considered as "the applied mathematics for modeling and analyzing systems that are based on information and communication technology " [32]. Their main objective is to establish system correctness with mathematical rigour. Their great potential has led to an increasing use by engineers of formal methods for the verification of complex systems.

Many institutions like the European ESA (European Space Agency) and the NorthAmerican NASA (North-Atlantic Space Agency) report that the formal verification techniques are the best way to avoid errors. During the last decade, research in formal methods has led to the development of powerful software tools that can be used tu automate various verification steps. A examples of this

kind of tools is TPAL [31, 49, 43] and UPPAAL which can model, simulate and verify developed systems.

Two kinds of formal verification approaches can be distinguished: deductive an model-based methods.

- In *deductive* methods, the correctness of systems is determined by properties in a mathematical theory, for example: theorem provers and proof checkers.
- The *model-based* techniques are based on models describing the possible system behaviour in a mathematical precise and unambiguous manner. However, *"Any verification using model-based techniques is only as good as the model of the system."* One of the most well-known an practically used verification techniques is *Model-Based Simulation*. The software tool, the simulator, allows the user to study the system behaviour. This technique is useful for a first, quick assessment or the quality or the design.

1.2.2 Model Checking

Model checking is a verification technique that explores all possible system states in a brute force manner [17, 18]. Similar to a computer chess program that checks possible moves, a model checker, the software tool that performs the model checking, examines all possible system scenarios in a systematic manner. In this way, it can be shown that a given system model truly satisfies a certain property. It is a real challenge to examine the largest possible state spaces that can be treated with current means, i.e., processors and memories.

Typical properties that can be checked using model checking are of a qualitative nature: Is the generated result ok?, Can the system reach a deadlock situation, e.g., when two concurrent programs are mutually waiting for each other an thus halt the entire system? But also timing properties can be checked: Can a deadlock occur within 1 hour after a system reset?, or Is a response always received within 8 minutes?

The system model is usually automatically generated from a model description that is specified in some appropriate dialect of programming languages like C or Java or hardware description languages

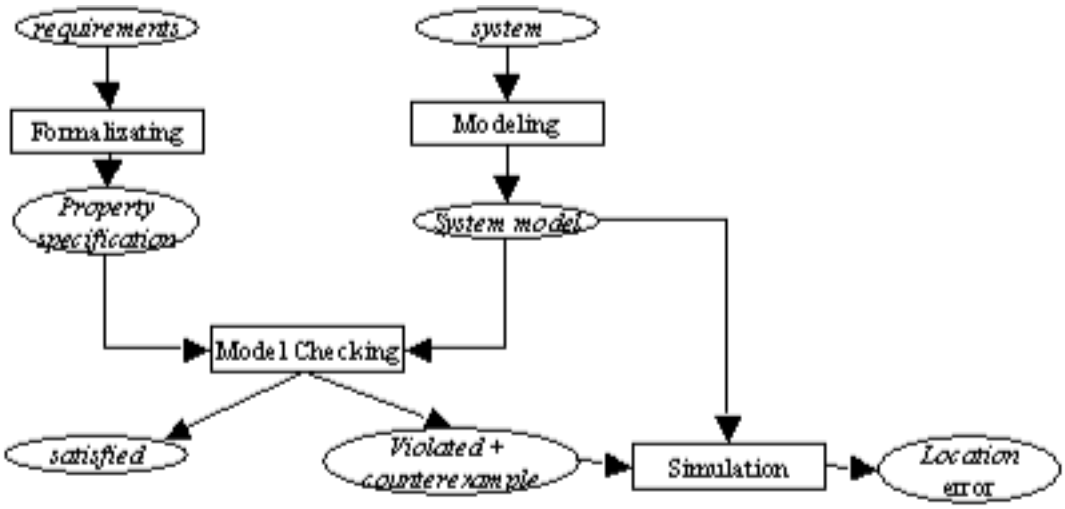

Figure 1.1: Schematic view of the model-checking approach

such as Verilog or VHDL. The model checker examines all relevant systems states to check whether they satisfy the desired property. If a state is encountered that violates the property under consideration, the model checker provides a counterexample describes an execution path that leads from the initial system state to a state that violates the property being verified. With the help of a simulator, the user can replay the violating scenario, in the way obtaining useful debugging information, and adapt the model or the property accordingly, cf. Figure 1.1.

1.2.3 Theorem Proving

The verification problem is interpreted as a mathematical theorem that typically has the form: *system specification* $\Rightarrow$ *desired property*. Trying to establish this result is referred to as *theorem proving*. In order to apply theorem proving, it is a prerequisite that the system specification has the form of a mathematical theory, or should be transformable into such form. Using a set of axioms , a theorem proveer tries to either construct a proof of the theorem by generation the intermediate proof steps, or to refuse it. The axioms are either built-in or are provided by the user. Theorem provers are also called *proof assistants*. The general demand to prove theorems of a rather general type and the use of undecid-

able logics requires some user interaction. Different variants exist: highly automated, general-purpose proof assistants, and interactive programs with special-purpose capabilities.

1.3 Characteristics of Model Checking

The principle of model checking is as follow [32] :

> Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.

This section treat the development of model checking and the advantages and drawbacks.

1.3.1 The development of Model Checking

Applying model checking to a design consists of several tasks, each of which will be discussed in detail.

Modeling The first task is to convert a design into a formalism accepted by a model checking tool. In many cases, this is simply a compilation task. In other cases, owing to limitations on timed and memory, the modeling of a design may require the use of abstraction to eliminate irrelevant or unimportant details.

Specification Before verification, it is necessary to state the properties that the design must satisfy, it is common to use *temporal Logic*, which can assert how the behavior of the system evolves over time. An important issue in specification is *completeness*. Model checking provides means for checking that a model of the design satisfies a given specification, but it is impossible to determine whether the given specification covers all the properties that the system should satisfy.

Verification Ideally the verification is completely automatic. However, in practice it often involves human assistance. One such manual activity is the analysis of the verification results. In case of a negative result, the user is often provided with an error trace. This can be used as a counterexample for the checked property and can help the designer in tracking down where the error occurred. In this case, analyzing the

error trace may require a modification to the system an reapplication of the model checking algorithm.

An error trace can also result form incorrect modeling of the system or from an incorrect specification. The error trace can also be useful in identifying and fixing these two problems. A final possibility is that the verification task will fail to terminate normally, due to the size of the model, which is too large to fit into the computer memory. In this case, it may be necessary to redo the verification after changing some or the parameters of the model checker or by adjusting the model.

1.3.2 Temporal Logic and Model Checking

Before, we consider the advantages and drawbacks of model checking, we will explain in more detail temporal logic and their importance to model checking [17, 18].

Temporal logics have proved to be useful for specifying concurrent systems, because they can describe the ordering of events in time without introducing time explicitly. They were originally developed by philosophers for investigating the way that time is used in natural language arguments. Although a number of different temporal logics have been studied, most have an operator like Gf that is true in the present if f is always true in the futures. To assert that two events e_1 and e_2 never occur at the same time, one would write $G(\neg e_1 \vee \neg e_2)$. Temporal logics are often classified according to whether time is assumed to have a *linear* or *branching* structure.

The introduction of temporal-logic model checking algorithms in the early 1980s allowed this type of reasoning to be automated. Because checking that a single model satisfies a formula is much easier than proving the validity of a formula for all models, it was possible to implement this technique very efficiently.

An example of a temporal logic is CTL^* is a very expressive logic that combines both branching-time and linear-time operators [16].

1.3.3 Advantages and Drawbacks

The advantages of model checking.

1. It is a *general* verification approach that is applicable to a wide range of applications such as embedded systems, software engineering and hardware design.

2. It supports *partial* verification, i.e., properties can be checked individually, thus allowing to focus on the essential properties first. No complete requirement specification is needed.

3. It is not vulnerable to the lidelihood with which an error is exposed; this contrasts with testing and simulation that are aimed at tracing the most probable defects.

4. It provides *diagnostic information* in case a property is invalidated; this is very useful for debugging purposes.

5. It is a potential "push-button" technology; the use of model checking does neither require a high degree of user-interaction nor a high degree of expertise.

6. It enjoys rapidly increasing *interest by industry;* several hardware companies started their in-house verification labs, job offers with required skills in model checking frequently appear, and commercial model checkers become available.

7. It can be easily *integrated* in existing development cycles; its learning curbe is not very steep, an empirical studies indicate that it may lead to shorter development times.

8. It has a *sound and mathematical underpinning*; it is based on elementary theory in graph algorithms, data structures, and logic.

The drawbacks of model checking.

1. It is mainly appropriate to *control-intensive* applications as data typically rangers over infinite domains.

2. It is applicability is subject to *decidability issues;*for infinite-state systems, or reasoning about abstract data types, model checking is in general not effectively computable.

3. It verifies a *system model,* and not the actual system itself; any obtained result is thus as the system model. Techniques such as testing are needed to find fabrication faults or coding errors.

4. It checks only *stated requirements,* i.e., there is no guarantee of completeness. The validity of properties that are not checked cannot be judged.

5. It suffers from the *state-space explosion* problem, i.e., the number of states needed to model the system accurately may easily exceed the amount of available computer memory. Despite the development of several very effective methods to combat this problem, models of realistic systems may still be too large to fit in memory.

6. It is usage requires some *expertise* in finding appropriate abstractions to obtain smaller system models and to state properties in the logical formalism used.

7. It is not guaranteed to yield correct results: as any tool, a model checker may contain *software defects.*

8. It does not allow to check *generalizations*: in general, checking systems with an arbitrary number of components, or parameterized systems can not be treated. Model checking can, however, suggest results for arbitrary parameters that may be verified using proof assistants.

These drawbacks suggest the conclusion:

Model checking is an effective technique to expose potential design errors.

Thus, model checking increases the correctness of a system design.

The next chapter treats a concrete tool named UPPAAL that is based on Model Checking.

Chapter 2

UPPAAL

A Tool for Automatic Verification of Real-Time Systems

UPPAAL is a tool suite for automatic verification of safety and bounded liveness properties of real-time systems modeled as networks of timed automata [37, 38, 36, 7, 5]. The UPPAAL engine transforms a certain class of linear hybrid systems to networks of timed automata, and implements techniques based on constraint-solving. UPPAAL also supports diagnostic model-checking providing diagnostic information in case verification of a particular real-time systems fails.
The current version of UPPAAL is available at **http://www.uppaal.com**.
The tool was developed during the spring of 1995 but, nowadays it is being extended with many features as distribution, guided, parameterized, cost-optimal, hierarchical (UML) or probability.

Next chapter is centered in the study of the probabilistic behaviour of UPPAAL models. But first we will to explain the basis of UPPAAL.

2.1 Real-Time Systems

UPPAAL use *time transition systems* as a basic semantical model for real-time system. The type of systems treated by UPPAAL is a

particular class of timed transition systems that are syntactically described by ***networks of timed automata*** [33, 51].

A timed transition system is a labelled transition system with two types of labels: atomic actions and delay actions (i.e. positive reals), representing discrete and continuous changes of real-time systems.

Let Act be a finite set of actions ranged over by a,b, etc., and P be a set of atomic propositions ranged over by p,q, etc. We use R to stand for the set of non-negative real numbers, Δ for the set of delay actions $\{\epsilon(d)|d \in R\}$ and L for the union $Act \bigcup \Delta$.

Definition 1. *A timed transition system over actions Act and atomic propositions P is a tuple $S = \langle S, s_0, \rightarrow, V\rangle$, where S is a set of states, s_0 is the initial state, $\rightarrow \subseteq S \times L \times S$ is a transition relation, and $V : X \rightarrow 2^P$ is a proposition assignment function.*

Note that the above definition is standard for labelled transition systems except that we introduced a proposition assignment function V, which for each state $s \in S$ assigns a set of atomic propositions $V(s)$ that hod in s.

In order to study compositionality problems we introduce a parallel composition between timed transition systems. Following [26] we suggested a composition parameterized with a synchronization function generalizing a large range of existing notions of parallel compositions. A ***synchronization function*** f is a partial function $(Act \bigcup \{0\}) \times (Act \bigcup \{0\}) \hookrightarrow Act$, where 0 denotes a distinguished no-action symbol. Now, let $\mathbf{S}_i = \langle S_i, s_{i,0}, \rightarrow_i, V_i\rangle$, $i = 1, 2$, be two timed transitions systems and leg f be a synchronization function. Then the parallel composition $\mathbf{S}_1|_f\mathbf{S}_2$ is the timed transition system $\langle S, s_0, \rightarrow, V\rangle$ where $s_1|_f s_2 \in S$ whenever $s_1 \in S_1$ and $s_2 \in S_2$, $s_0 = s_{1,0}|_f s_{2,0} \rightarrow$ is inductively defined as follows:

- - $s_1|_f s_2 \xrightarrow{c} s'_1|_f s^1{}_2 if s_1 \xrightarrow{a}_1 s'_1, s_2 \xrightarrow{b}_2 s^1{}_2$ and $f(a, b) = c$
- - $s_1|_f s_2 \xrightarrow{\in(d)} s'_1|_f s^1{}_2 if s_1 \xrightarrow{\in(d)}_1 s'_1, s_2 \xrightarrow{\in(d)}_2 s^1{}_2$

and finally, the proposition assignment function V is defined by $V(s_1|_f s_2) = V_1(s_1) \bigcup V_2(s_2)$.

Note also that the set of states and the transition relation of a timed transition system may be infinite. We shall use networks of timed automata as a finite syntactical representation to describe timed transition systems.

2.2 The UPPAAL technique for modeling Real-Time Systems

In this section, we study real-time systems consisting of communicating processes with shared clocks. The systems are described by networks of timed automata [2] extended with auxiliary data variables and with a notion of parallel composition. Instead of interpreting parallel composition as logical conjunction we use a CCS-like [46] interpretation of parallel composition (proposed in [51]), allowing one-to-one communication and interleaving.

2.2.1 Networks of Timed Automata

By definition, a timed automata is a standard finite-state automata extended with a finite collection of real valued clocks. The clocks are assumed to proceed at the same rate and their values may be compared with natural numbers or reset to 0. We have extended the notion of timed automata to include integer variables, i.e. integer valued variables that may be compared to natural numbers or assigned to any value of the form $ax + b$ where $a, b \in \mathbf{Z}$ and x is the variable being reassigned.

The model also allows clocks not only to be reset, but also to be set to any non-negative integer value.

Definition 2. (Atomic Constraints) *Let C be a set of real valued clocks an I a set of integer valued variables. An atomic clock constraint over C is a constraint of te form: $x \sim n$, for $x \in C$, $\sim \in \{\leq, \geq, =\}$ and $n \in \mathbf{N}$. An atomic constraint over I is a constraint over I is a constraint of the form: $i \sim n$, for $i \in I$, $\sim \in \{\leq, \geq, =\}$ and $n \in \mathbf{Z}$*

Let $C_c(C)$ denote the set of all clock constraints over C, and let $C_i(I)$ denote the set of all integer constraints over I.

Definition 3. Guards *Let C be a set of real valued clocks and I a set of integer valued variables. A guard g over C and I is a formula generated by the following syntax: $g ::= c | g \wedge g$, where $c \in (C_c(C) \bigcup C_i(I))$*

We let $\mathcal{B}(C, I)$ stand for the set of all guards over C and I

Definition 4. Assignments *Let C be a set of real valued clocks and I a set of integer valued variables. A clock assign over C is a tuple $\langle v, c \rangle$, where $v \in C$ and $c \in \mathbf{N}$. An integer assign over I is a tuple $\langle v, c_1, c_2 \rangle$ representing the assignment $v = C_1 \cdot v + c_2$, where $v \in I$ and $C_1, c_2 \in \mathbf{Z}$.*

We will use $\mathcal{A}(C, I)$ to denote the power-set of all assignments over I and C.

Definition 5. Timed automata *A timed automata A over a finite set of actions Act, clocks$C$ and integer variables I is a tuple $\langle L, l_0, E \rangle$ where L is a finite set of nodes (control-nodes), l_0 is the initial node, and $E \subseteq L \times \mathcal{B}(C, I) \times Act \times \mathcal{A}(C, I) \times L$ corresponds to the set of edges. To denotes, $\langle l, g, a, r, l' \rangle \in E$, we will write $l \xrightarrow{g,a,r} l'$.*

In order to study compositionality problems we introduce a parallel composition between timed automata. In order to get the kind of parallel composition we want, we have to introduce the notion of co-actions, that is done by defining a synchronization function $\mathcal{T}$.

Definition 6. Synchronization Function *Let $\mathcal{T} \subseteq Act \times Act$ be a function such that:*

$$\langle a_i, a_j \rangle \in \mathcal{T} \Rightarrow \langle a_j, a_i \rangle \in \mathcal{T} \text{ for all } a_i, a_j$$

Definition 7. Parallel Composition *Let A_1, A_2 be two timed automata. Then the parallel composition $(A_1|A_2)$ is a timed automata $\langle L, l_0, E \rangle$ where $(l_1|l_2) \in L$ whenever $l_1 \in L_1$ and $l_2 \in L_2$, $l_0 = (l_{1,0}|l_{2,0})$. The edges E are defined as follows:*

- $(l_1|l_2) \xrightarrow{g,\tau,r} (l_1'|l_2')$ *if* $(l_1 \xrightarrow{g_1,a_1,r_1} l_1') \wedge (l_2 \xrightarrow{g_2,a_2,r_2} l_2') \wedge (g = g_1 \cup g_2) \wedge (\langle a_1, a_2 \rangle \in \mathcal{T}) \wedge (r = r_1 \cup r_2)$
- $(l_1|l_2) \xrightarrow{g,a,r} (l_1'|l_2)$ *if* $(l_1 \xrightarrow{g,a,r} l_1')$
- $(l_1|l_2) \xrightarrow{g,a,r} (l_1|l_2')$ *if* $(l_2 \xrightarrow{g,a,r} l_2')$

Note that parallel composition is commutative and associative.

A states of a timed automata A is a pair $\langle l, u\rangle$ where l is a node de A and u is an assignment, mapping each clock in C to a value in $\mathbf{R}_+$, and each integer variable in I to a value in $\mathbf{Z}$. We will use $g(n)$ to denote that the assignment u satisfies the guard g. The initial state of A is $\langle l_0, u_0\rangle$, where u_0 is the assignment mapping all variables to 0.

An automata may take two types of transitions, from state to state:

- Delay transition: $\langle l, u\rangle \xrightarrow{\in(d)} \langle l, u'\rangle$ following the rules given in Definition 8.
- Action transition: $\langle l, u\rangle \xrightarrow{g,a,r} \langle l', u'\rangle$ following the rules given in Definition 9.

Definition 8. (Delay transition) *Let $\langle l, u\rangle$ and $\langle l', u'\rangle$ be two states of a timed automata A, and let d be a positive real. Then*

$$\langle l, u\rangle \xrightarrow{\in(d)} \langle l', u'\rangle \begin{cases} l' = l & \\ u'(x) = u(x) + d & \text{if } x \in C \\ u'(x) = u(x) & \text{if } x \in I \\ d \leq M(l, u) & \end{cases}$$

Where $M(l, u)$ is the maximal delay of $\langle l, u\rangle$ defined as follows:

$$M(l, u) = \begin{cases} sup\{t | g(u + t)\} & \text{if } \exists l' : l \xrightarrow{g,a,r} l' \\ \infty & \text{otherwise} \end{cases}$$

Intuitively this means that an automata may not stay in a contro-state long enough for the last outgoing edge from that state to close. This results in a maximal progress behavior of the automata.

Definition 9. (Action transition) *Let $\langle l, u\rangle$ and $\langle l', u'\rangle$ be two states of a timed automata A. Then*

$$\langle l, u\rangle \xrightarrow{g,a,r} \langle l', u'\rangle \text{ iff } \wedge g(u) \wedge \left(u'(x) = \begin{cases} c_0 & \text{if } x \in C \wedge \langle x, c_0\rangle \in r \\ c_1 u(x) + c_0 & \text{if } x \in I \wedge \langle x, c_1, c_0\rangle \in r \\ u(x) & \text{otherwise} \end{cases} \right)$$

2.2.2 Networks of Linear Hybrid Automata

The model of timed automata can be extended by allowing the clocks to proceed with different rates and also allowing the rates to drift within a bounded interval. This model is often referred to as linear hybrid automata.

It has been shown in [41] that for a restricted subclass of linear hybrid automata each linear hybrid automata can be simulated by a timed automata. The restrictions needed are:

- The clocks must never be stopped, i.e. it must never be possible for a clock to have rate 0.
- If we have an edge e between two nodes l and l', any clock which rate in l' differs from the rate in l must be reset on e.
- Any clock with negative rate must have a finite upper bound in each node, and any clock with positive rate must have a finite lower bound in each node.

2.3 Verifying Real-time Systems in UPPAAL

As mentioned in the introduction nowadays, the increasing use of systems which deal with time requirements, makes it necessary to develop methods to check system behaviour correctness.

As pointed out in [51, 23], the practical goal of verification of real-time systems, is to verify simple safety properties such as "can we guarantee that a bad thing won't occur?" or "are we sure that eventually a good thing will occur?". These properties could be formalized in temporal logic as $\forall\Box\neg bad - thing$ and $\exists\Diamond good - thing$ In finite-state systems this kind of properties can be verified by checking all possible reachable states of a system. But the systems considered now are infinite-state because of the clocks are real-valued.

2.3.1 Symbolic Model-Checking

The region-graph technique pointed out in [23, 51] allows the state space of a real time system to be partitioned into finitely

many regions in such a way that states within the same region satisfy the same properties. However, as the notion of region is essentially property-independet and the number of such regions depends highly on the constants used in the clocks constraints of an automata, the region partitioning is extremely fine (and large). This section offers a much coarser (and smaller) partitioning of the states space yielding extremely efficient model-checking.

The semantical state of a network of timed automata is a pair (l, u) where l is a control-node and $u \in \mathbf{R}$ is a clock assignment in the form $\langle (l, u), v \rangle$ satisfy a given formula φ, that is,

$$\langle (l, u), v \rangle \models \varphi$$

Note that u is a clock assignment for the automata clocks and v is a clock assignment for the formula clocks. Now, the problem is that we have too many (in fact, infinitely many) such assignments to check in order to conclude $\langle (l, u), v \rangle \models \varphi$.

In this section, we shall use clock constraints $\mathcal{B}(C \cup K)$ for automata clocks C and formula clocks K to symbolically represent clock assignments. We shall use D to range over $\mathcal{B}(C \cup K)$. For safety formulas φ we show an algorithm to simultaneously check

$$[l, D] \models \varphi$$

which means that for each u and v such that uv is a solution to the constraint system D, we have $\langle (l, u), v \rangle \models \varphi$.

Thus the space $\mathbf{R}_{C \cup k}$ is partitioned in terms of clock constraints. As for a given network and a given formula, we have only finite many such constraints to check, the problem becomes decidable, and in fact as the partitioning takes account of the particular property, the number of partitions es in practice considerably smaller compared with the region-technique.

2.3.2 Reachability Analysis by constraint Solving

Adopting the methodology developed in [51], we start with describing a simple reachability problem for timed automata. A generalized version of the problem will be given later.

Definition 10. (Simple Reachability) *Let* $\langle l_0, u_0 \rangle$ *and* $\langle l_f, u_f \rangle$ *be states of a timed automata A. Then* $\langle l_f, u_f \rangle$ *is reachable from* $\langle l_0, u_0 \rangle$ *iff there exists a trace* $\langle l_0, u_0 \rangle \xrightarrow{Q_1} ... \xrightarrow{Q_n} \langle l_f, u_f \rangle$ *that leads to the final state.*

Note that the simple reachability relation is the transitive and reflexive closure of the transition relation.

To solve the problem with infinite state-space we use constraint systems to symbolically represent sets of assignments that we call time regions.

Definition 11. General Reachability *Let* l_0, l_f *be nodes of a timed automata A, and let* U_0, U_f *be sets of assignments. We say that* $\langle l_f, U_f \rangle$ *is reachable from* $\langle l_0, U_0 \rangle$ *iff there exists* $u_0 \in U_0$ *and* $u_f \in U_f$ *such that* $\langle l_f, u_f \rangle$ *is reachable from* $\langle l_0, u_0 \rangle$.

We will now describe an algorithm for solving the general reachability problem, using constraint solving techniques.

Let A **be the timed automata to be analyzed. We assume that the clocks of** A **can be ordered as a vector** $\langle c_1, c_2, ..., C_n \rangle$**. Then the clock part of an assignment can be seen as a point in the n-dimensional space** $\mathbf{R}^n_{+c}$**. We will use linear constraint systems to describe regions of such points, and solve the general reachability problem by manipulating such linear constraint systems. A class of Simple linear Constraint Systems**

The class of constraints we will study will always take the form: $l_i \leq x_i \leq u_i$ **or** $l_{ij} \leq x_i - x_j \leq u_{ij}$**. we will use the term *linear constraint system* to denote a set of constraints of that form. A *solution* to a linear constraint system** r **is an assignment that maps each variable to a value that satisfies the constraints.**

The solutions set of a constraint system r **(i.e. the set of all solutions to r) can be seen as a convex polytope in the n-dimensional space. It can be seen in the figure 2.1.**

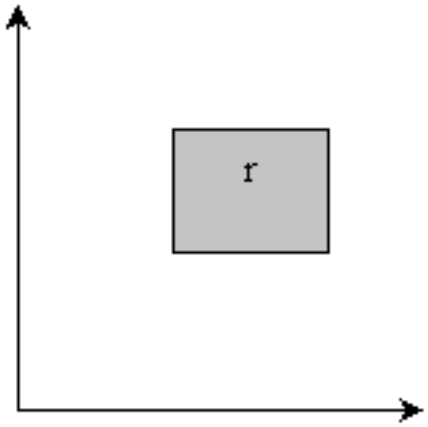

Figure 2.1: A solution region r

From now on we will simply call a constraint system r a *region* to its solution set. We will use $r = \emptyset$ to denote that r is not satisfiable, (i.e. its solution set is empty), $r \subseteq r'$ to denote that r implies r' and $r \wedge r'$ to denote the intersection of the solution sets of r and r'.

To represent a constraint system, we introduce a particular clock x_0 which always has value 0. Then we may assume that all constraints in the system have the form $l_{ij} \leq x_i - x_j \leq u_{ij}$. Constraint systems on this form can easily be represented by matrices.

2.3.3 Operations on Constraint Systems

The subject of this section is to describe an algorithm that decides if a symbolic state $\langle l_f, U_f \rangle$ is reachable from a symbolic state $\langle l_0, U_0 \rangle$.

The algorithm will manipulate r by use of constraint solving. The guards and resets on the transitions and delays in the state are the things that affect the constraints and motivate the operations introduced. The direction of the reachability analysis, forward or backward, will need different operations on constraint system.

Definition 12. the delay operations

1. *The weakest precondition of a timed region r is defined as*

$$wp(r) = \{x | \exists d \geq 0 : x + d \in r\}$$

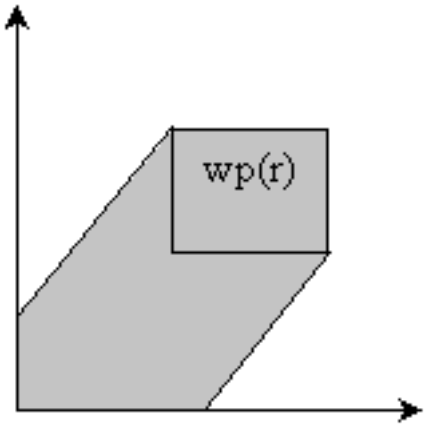

Figure 2.2: The operation weakest precondition over Constraint System r

2. *The* strongest postcondition *of a time region r is defined as*

$$sp(r) = \{x | \exists d \geq 0 : x - d \in r\}$$

This operations are needed because the automata may delay in a state, perform delay transitions, before an action transition is performed. If the reachability analysis is forward we start from $\langle l_0, U_0 \rangle$ and search forward for $\langle l_f, U_f \rangle$. In this case strongest postcondition is used. If we do backward reachability analysis we search backward from $\langle l_f, U_f \rangle$ to $\langle l_0, U_0 \rangle$ and handle delays with weakest precondition. The delay operations *weakest precondition* and *strongest precondition* are shown in the figures 2.2 and 2.3 respectively.

Definition 13. guard conjunction *If r is a time region and g is a set of guards the time region after the guard conjunction is the intersection of the solution sets: $g \cup r$*

This operation is performed when determining which transitions are enabled or not. If r' becomes inconsistent we have $r' = \emptyset$.

Definition 14. operations treating resets *Let r be a time region and k then clock to be reset.*

1. *$reset(r, k)$ is defined as*

$$\{v | \exists w \in r : v_k = 0 \wedge \forall i \neq k : v_i = w_i\}$$

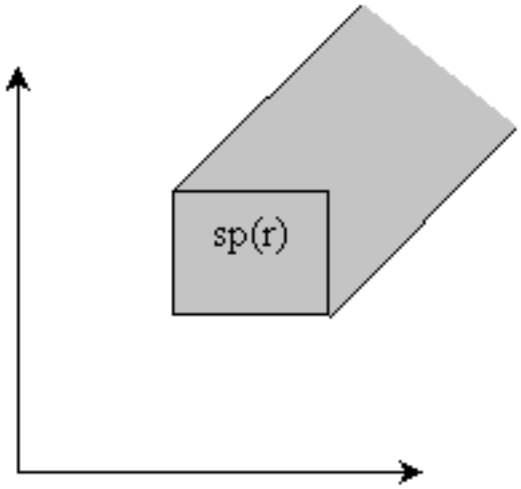

Figure 2.3: The operation strongest precondition over Constraint System r

2. *$free(r,k)$ is defined as*

$$\{v | \exists w \in r : v_k \in \mathbf{R} \wedge \forall i \neq k : v_i = w_i\}$$

3. *The Boolean operation $check(r,k)$ is defined as*

$$\{v | v \in r \wedge v_k = 0\} \neq \emptyset$$

The reset operation is used in forward reachability analysis to handle the resets that may be performed when the system perform a transition. In backward reachability analysis the free and check operations are used to handle the resets on the transitions. It is straight forward to extend the above definition to handle sets of resets and resets involving other values than 0. The operations *reset* and *free* are shown int the figures 2.4 and 2.5 respectively.

The implementation of the algorithm and operations are much easier if the time regions are of a special form called closed.

Definition 15. *Let r be a time region of the form $x_i - x_j \leq u_{i,j}, x_0 = 0$, r is closed if it satisfies*

$$\forall d_{i,j} \leq u_{i,j} \exists x_i \exists x_j : x_i - x_j = d_{i,j}$$

Time regions are closed under operations defined above.

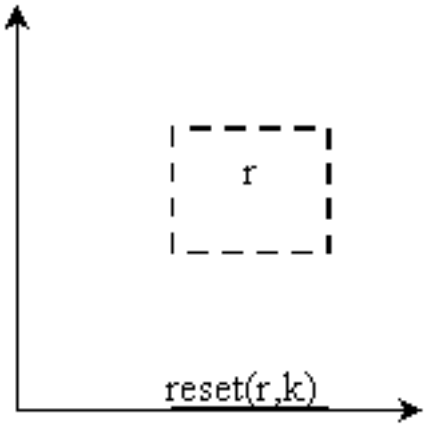

Figure 2.4: The operation reset over Constraint System r

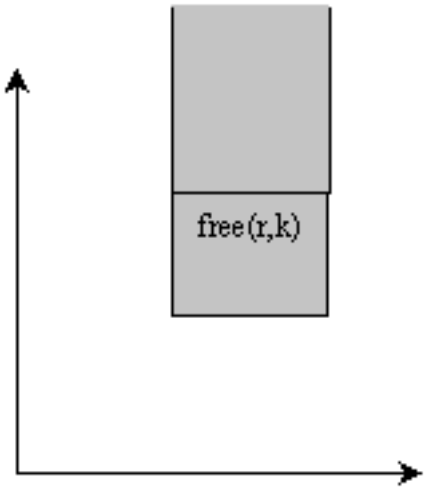

Figure 2.5: The operation free over Constraint System r

2.4 Algorithms

We're now in a position to present an algorithm reachability analysis using constraint solving. The algorithm checks if a state in a timed automata is reachable from the initial state or not.

When searching the state space we need two buffers that we can call wait and passed respectively. The wait buffer holds the states not wet explored and the passed buffer holds the states explored so far. We will treat the froward and backward reachability analysis algorithms in separate subsections.

2.4.1 Forward Reachability Analysis

If we do forward reachability analysis we initially store $\langle l_0, U_0 \rangle$ in the wait buffer. We then repeat the following:

Algorithm 1. *1. Pick a state $\langle l_i, U_i \rangle$ from the wait buffer.*

2. *Check if $l_i = l_f \wedge U_i \subseteq U_f$. If that is the case, return the answer yes.*
3. *if $l_i = l_j \wedge U_i \subseteq U_j$, for some $\langle l_i, U_i \rangle$ in the passed buffer, drop $\langle l_i, U_i \rangle$ and go to step 1. Otherwise save $\langle l_i, U_i \rangle$ in the passed buffer. If $U_j \subset U_i$ we can replace the state $\langle l_j, U_j \rangle$ with $\langle l_i, U_i \rangle$. (To save space)*
4. *Find all l_k that are reachable from l_i in one step regardless of the assignments, taking only actions into account. Let g_k be the set of guards on the performed transition an a_k the set of resets*
5. *Now set $U_k = reset(sp(U_i) \cap g_k, a_k)$. If $U_k \neq \emptyset$, store $\langle l_k, U_k \rangle$ in the wait buffer.*
6. *if the wait buffer is not empty go to step 1, otherwise return the answer no.*

2.4.2 Backward Reachability Analysis

To simply the algorithm description below we define an operation *inv_reset* which entirely is composed of operations defined above.

Definition 16. *Let r be a time region and a a set of resets. We define*

$$inv_reset(r, a) = \begin{cases} free(r, a) & check(r, a) \\ \emptyset & otherwise \end{cases}$$

If we do backward reachability analysis we initially store $\langle l_f, U_f \rangle$ in the wait buffer. We then repeat

Algorithm 2. *1. Pick a state $\langle l_i, U_i \rangle$ from the wait buffer.*

2. *Check if $l_i = l_0 \wedge U_i \subseteq U_0$. If that is the case, return the answer yes.*
3. *if $l_i = l_j \wedge U_i \subseteq U_j$, for some $\langle l_i, U_i \rangle$ in the passed buffer, drop $\langle l_i, U_i \rangle$ and go to step 1. Otherwise save $\langle l_i, U_i \rangle$ in the passed buffer. If $U_j \subset U_i$ we can replace the state $\langle l_j, U_j \rangle$ with $\langle l_i, U_i \rangle$. (To save space)*
4. *Find all l_k that leads to l_i in one step regardless of the assignments, taking only actions into account. Let g_k be the set of guards on the performed transition an a_k the set of resets.*
5. *Now set $U_k = inv_reset(wp(U_i) \cap g_k, a_k)$. If $U_k \neq \emptyset$, store $\langle l_k, U_k \rangle$ in the wait buffer.*
6. *if the wait buffer is not empty go to step 1, otherwise return the answer no.*

2.5 A overview of the UPPAAL GUI

UPPAAL [37, 36, 7] main window (figure 2.6) has two main parts: the menu and the tabs.The menu is described in the integrated help, accessible though the help menu. This menu describes the GUI in details, so this section will rather focus on how to use it. The three tab panels give access to the three components of UPPAAL that are the editor, the simulator and the verifier. Figure 2.6 shows the editor view. The idea is to define a bunch of templates (like in C++) that are instanciated to have a complete system. The templates can have symbolic variables and constants as parameters and they may have local variables as well. To get a first contact with UPPAAL, double click in the drawing area to get a location, repeat this, you have two. Double click on these locations to rename them to start and end. Click on the Transition Mode, click on the start location and on the end location. You have your first automata ready, as depicted in figure 2.7. Click on the Simulator tab to start the simulator, click on the yes button that will popup and you are ready to run your first system. Figure 3 shows the simulator view. On the left you will find the control part where you can choose the transitions (upper part) and replay/save/load a trace (lower part). In the middle are the variables and on the right the system itself.

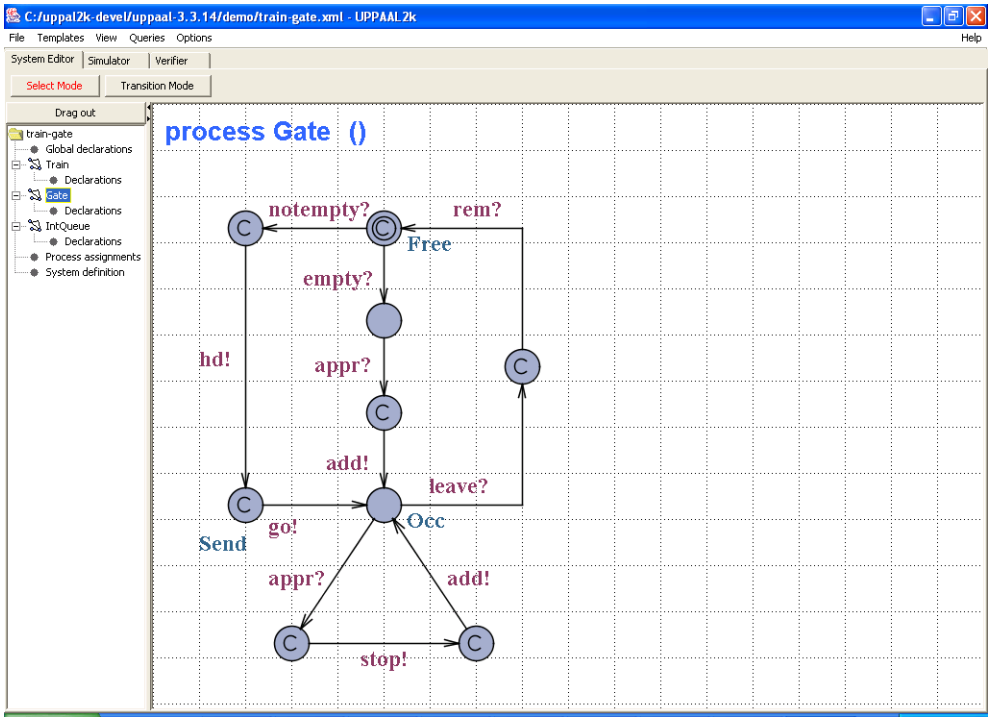

Figure 2.6: Overview of UPPAAL

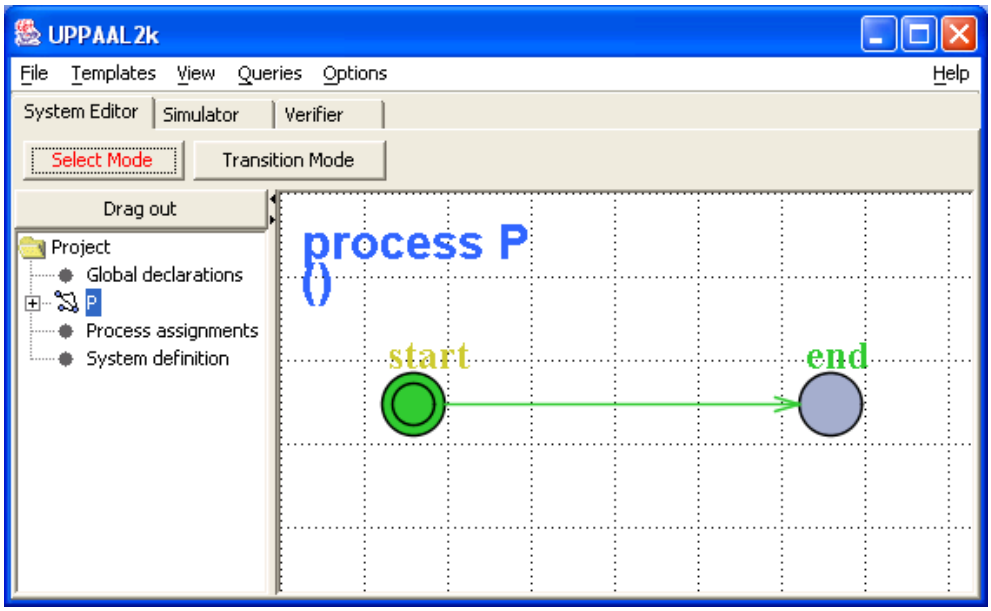

Figure 2.7: First model

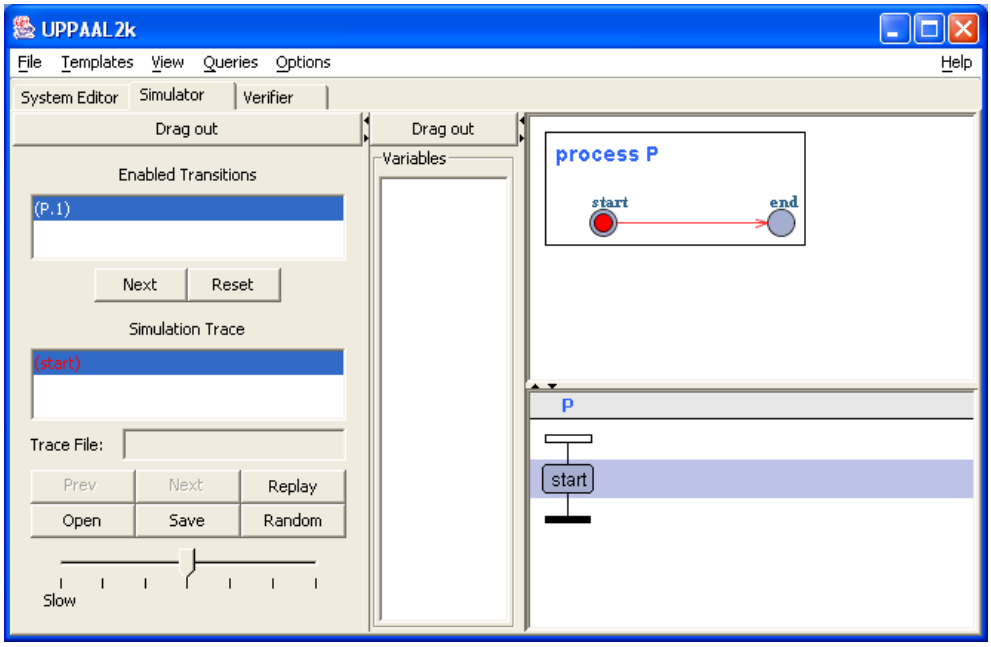

Figure 2.8: A snapshot of the simulator

The last tab is the Verifier which is shown in the figure 2.9. It is divided in two parts.At the top part you can describe and sound the properties and the bottom part shows the connection status with the local verifier server which is the real verifier. The top part is divided in three panels. At the top, the Overview panel allows us to see all the properties to be sound (it has two modes properties and comments). The middle panel is the property one which allows us to add new properties, and the bottom panel is the comments one which allows us to write the property comments. On the right you can see some buttons which allows us to check the properties, change the mode between property view or comment view and, remove and insert new properties. All of this can be seen in the figure 2.9.

There is a small tutorial about UPPAAL for the reader available at the URL:"http://www.docs.uu.se/docs/rtmv/uppaal/tutorial.pdf" and several study cases in [25, 24, 21, 28, 1, 39, 9, 10, 34, 35, 6, 40].

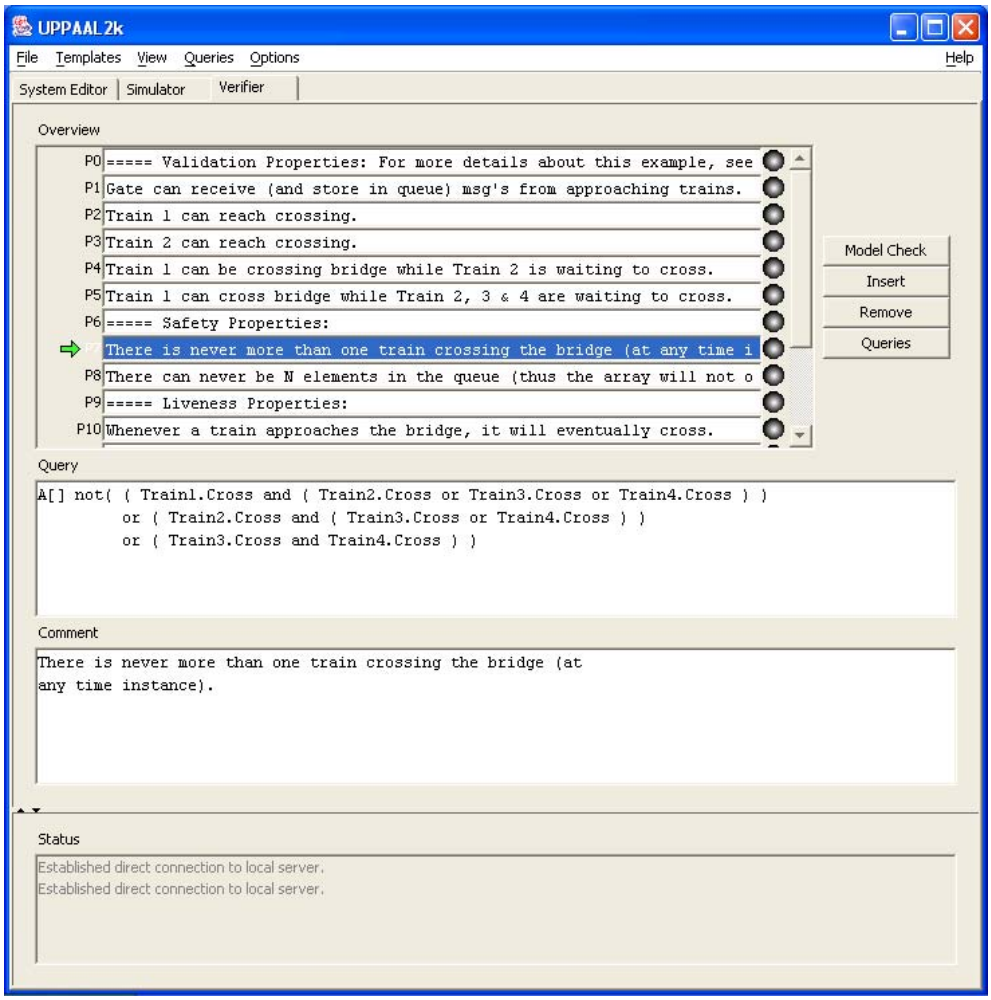

Figure 2.9: A snapshot of the verifier

In the next chapter we will see the suite tool P_UPPAAL and we will prove the parallel behaviour between UPPAAL and P_UPPAAL.

Chapter 3

The suite tool P_UPPAAL

In this chapter we treat about the suite tool P_UPPAAL in detail. We show the P_UPPAAL functionality (system modeling, simulation and verification) and its internal behaviour. To see the P_UPPAAL functionality, this chapter describes its graphical interface. And to see the P_UPPAAL internal behaviour, it shows the tool named RAPTURE.

3.1 Introduction

The suite tool P_UPPAAL allows us the system modeling, simulation and verification. The simulation process is required for validating the system modeling. The modeling and simulation processes are led through graphical interfaz and the verification process is realized by RAPTURE [27, 20, 19]. This tool, P_UPPAAL, introduces the probabilistic behaviour to the UPPAAL models. To add this behaviour, the tool imports models from UPPAAL and translates them into a graphical model. The graphical model is translated to a textual format which is used by the RAPTURE engine.

The Rapture engine provides the strategies for model checking quantitative reachability properties of Markov decision processes [45] by successive refinements. The properties are analyzed on abstractions rather than directly on the given model. Such abstractions are expected to be significantly smaller than the original model, and may safely refute or accept the required property. Otherwise, the abstraction is refined and the process repeated. As the

numerical analysis involved in settling the validity of the property is more costly than the refinement process, the method profits from applying such numerical analysis on smaller state spaces.

Model checking of finite state systems allows settlement of qualitative properties such as "the system will never reach an erroneous situation". However, it is often vital that additional quantitative properties are established in order for the system to be considered correct. Such properties include real-time requirements such as "a desired state will be reached within 105 seconds" and probabilistic properties of the type "a desired state will be reached with probability at least 99% ".

Thus, this kind of Model checking allows us to introduce the probabilistic behaviour into UPPAAL models.

Notice that, P_UPPAAL does not require UPPAAL to work. Because it allows modeling the system using the editor provided and after that, it export the system modeled to a UPPAAL model. Furthermore, it might validate the system modeled and verify quantitative reachability properties within the system modeled. Thus, P_UPPAAL is a suite of tools.

3.2 P_UPPAAL Architecture and Functionality

The architecture of P_UPPAAL, shown in figure 3.1, is divided in threw main parts, i.e. the Editor, the Simulatior and the Verifier.

The Editor allows us to model the system studied. The system is modeled as a probability transition system (PTS for short). The probability behavior of the system is captured by a variable in the transitions.

The Simulator allows us to validate the system. Through the simulation, you may decide if the system modeled holds the behaviour expected.

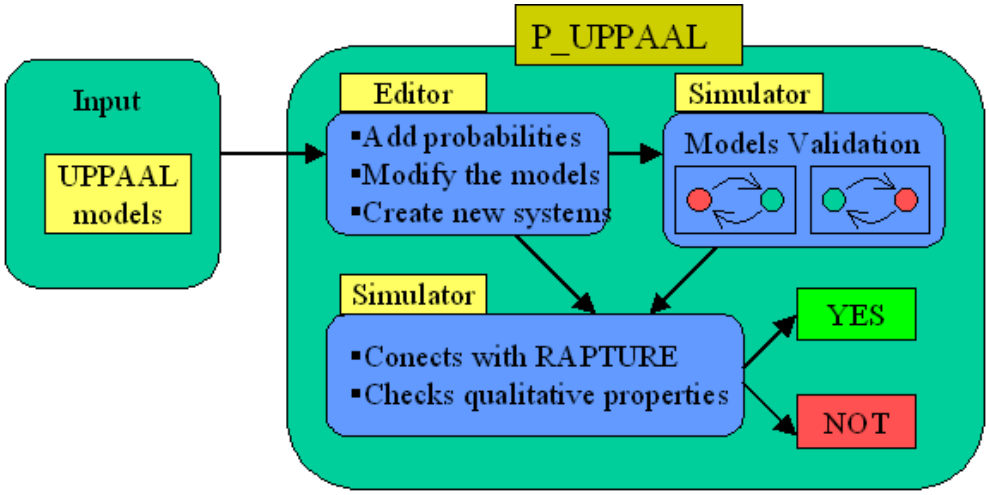

Figure 3.1: Architecture of the RAPTURE tool

The verifier checks the quantitative reachability properties defined on it. It translates the system modeled into textual format and, after that, it connects with RAPTURE tool which delivers the verdict. Therefore, if the system holds the property defined, the verifier receives from RAPTURE the answer yes, otherwise, it receives the answer not.

Now we treat all of this into detail.

3.3 Importing Models from UPPAAL

The import process is based in the following idea: "The UPPAAL and P_UPPAAL models are derived from Labeled Transitions System". It allow us to abstract the modeling phase. Thus, we only import from the UPPAAL models the states, the transitions, the variables and channels, the synchronization, the guards; i.e, We import all the model except the timing characteristics (invariants and clocks).

The figures 3.2 and 3.3, shows a simple example of the original model from UPPAAL and the imported model in P_UPPAAL respectively. As you can see in the figure 3.3, the import model does not have the clock i which could be seen at the UPPAAL model.

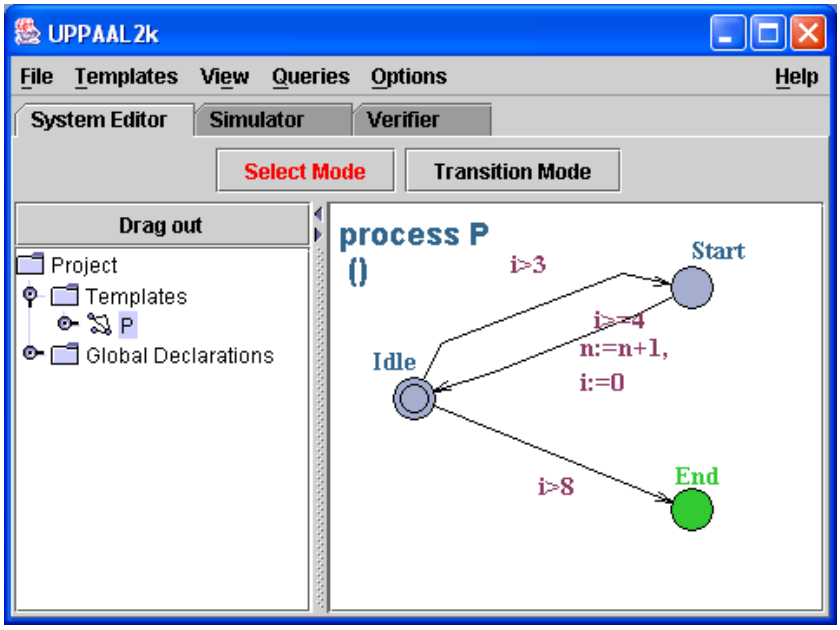

Figure 3.2: An example of UPPAAL model.

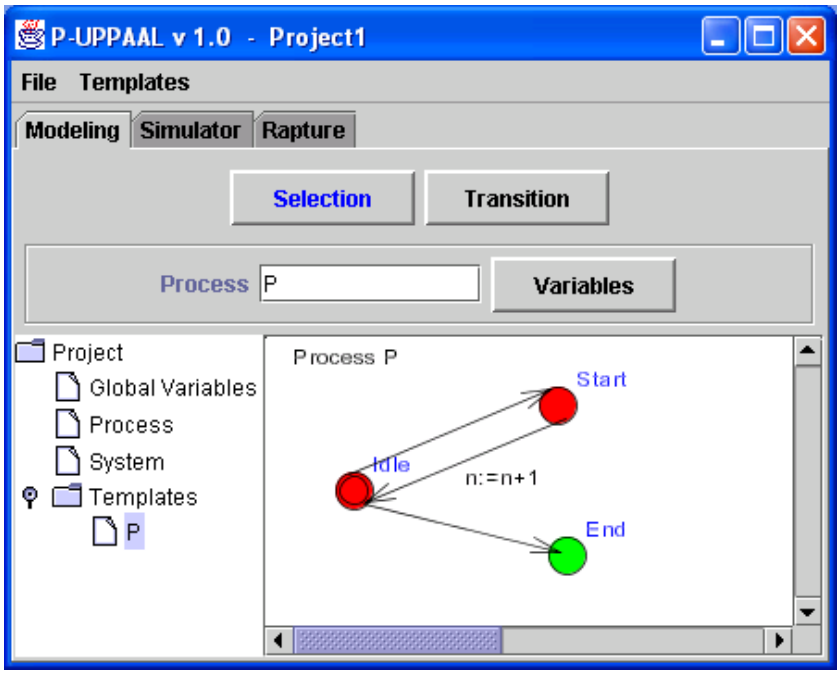

Figure 3.3: The P_UPPAAL imported model from UPPAAL.

You may found the *import option* at the P_UPPAAL menu that opens a dialog to search the UPPAAL model for import process.

3.4 Modeling Systems

P_UPPAAL provides three tabs: Editor, Simulator y RAPTURE. The first tab, the Editor, is a graphical interface to draw the model. This graphical interface allows us to describe and draw the probabilistic transition system.

The Editor is divided in two main panels:

- *The left panel* allows us to describe the processes, the variables, the system and the templates, that will be part of the model. In **variables**, you can define integral variables and channels that provide the synchronization between the processes. In **process**, you can define the process from the different templates defined before. Finally, in **system**, you can describe the processes that run in parallel.
- *The right panel* allows us to draw the template as a PTS. Thus, we can draw states (initial or not) and transitions and its guards, synchronizations, assignments and probabilities.

The editor provides two modes to work, the selection mode and the transition mode. *The selection mode* allows us to create new states or erase it. Furthermore, it allows us to edit the state and transition properties; i.e, it edits the name of the state and it checks if the state is initial, and, it edits the transitions properties: guards, assignments, synchronizations between other processes through the channels and probabilities. *The mode transition* allows us to draw the relations that connect the states with transitions, but we have to change to the selection mode for add the transitions properties seen before.

In the figure 3.4 , we can see the tab Editor selected with a simple model. This model has been only designed with one process. First, it has three states: Idle, Start and End. The probability to reach the state End is only *0.1*. While, the probability to reach the state Start is only *0.9*. This example is the model imported from UPPAAL (figure 3.2) modified with transition probabilities and others changes.

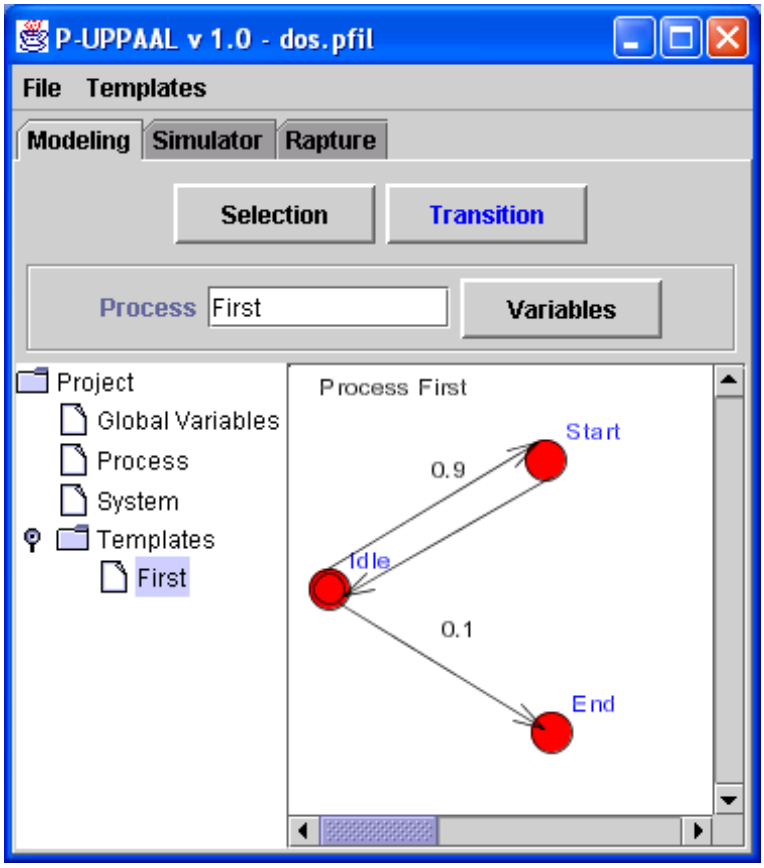

Figure 3.4: A P_UPPAAL model example.

3.5 Systems Simulator

The P_UPPAAL tab Simulator provides a way to validate the system under study. We can check if the system behaviour is correct, i.e., if the system holds all the design requirements that we have collected previously.

To check the model syntaxis, P_UPPAAL uses a parser in JAVA. The simulation algorithm used by P_UPPAAL is as follows:

1. Checks the guards.
2. Store the enabled transitions, if enabled transitions $= \emptyset \Rightarrow$ Deadlock (The simulation end)
3. Choose the transition or transitions (if there are synchronization).
4. Execute transition $\Rightarrow$
   - Make the assignments.
   - Make the synchronization.
5. Return to the beginning.

The simulator panel is divided in four different panels: enabled transitions, simulator trace, variables and the system. The panel "Enabled Transitions" collects the transitions that are enabled now. The panel "Simulation Trace" shows the trace history, i.e., It shows the transitions that have been simulated before. The panel "Variables" shows all the systems variables and its current value. The last panel shows the running simulation.

Before runing a simulation, you have to save the system modeled. After that, you can start the simulation. The first step is to choose the transition to run, it is made from the panel "Enabled transition". Then you push Next button and the generated trace is stored in the trace panel.

The figure 3.5 shows the simulator running a simulation for the system shown in the figure 3.4.

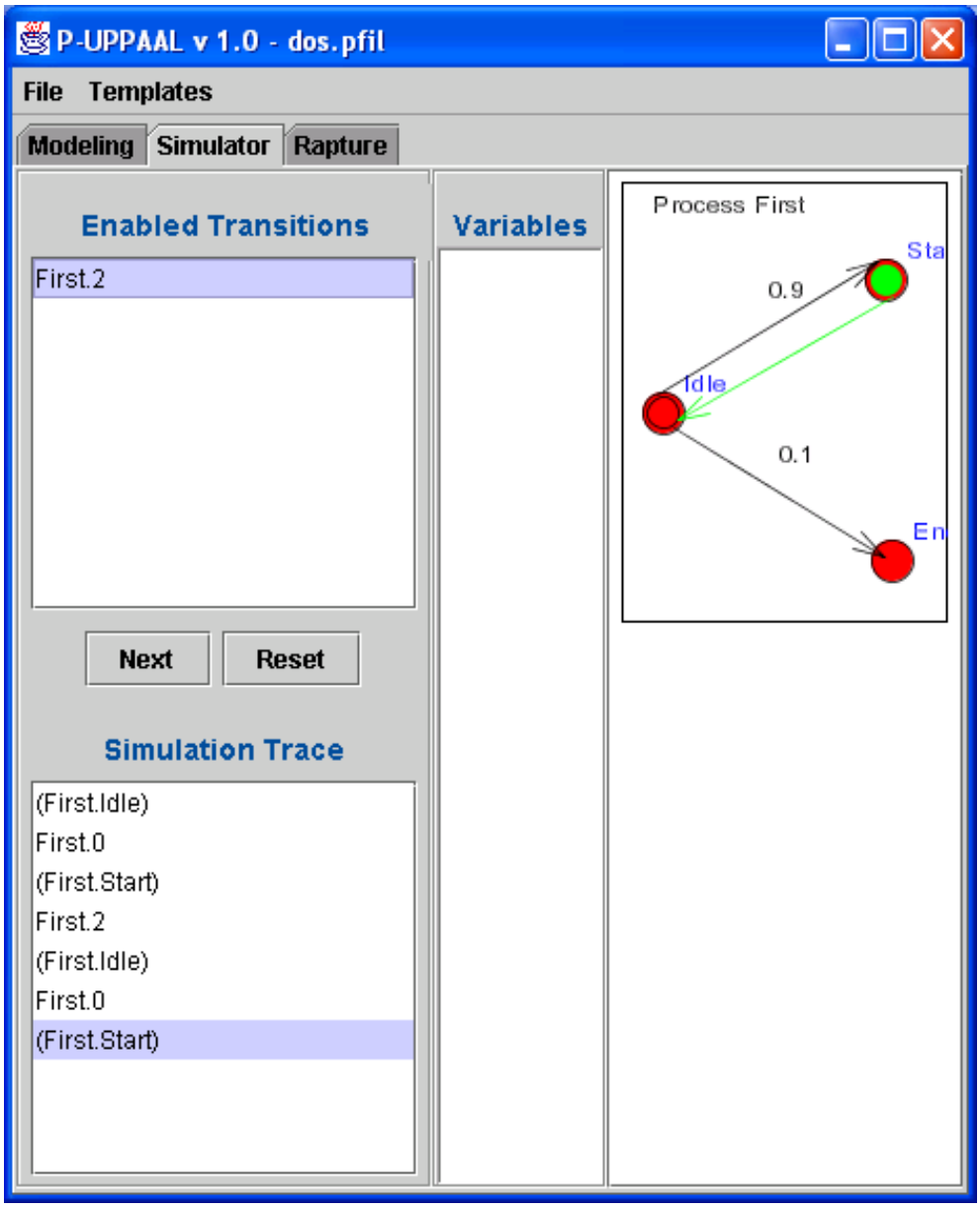

Figure 3.5: A simulation example.

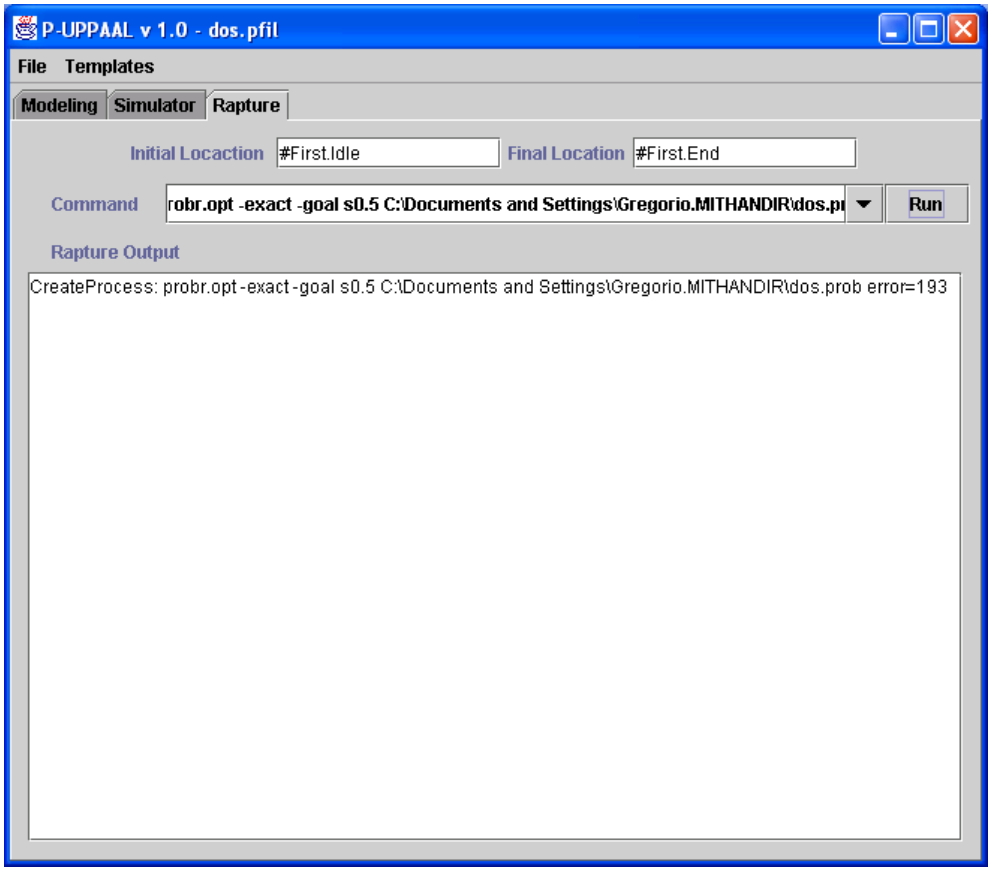

Figure 3.6: System verification.

3.6 The Verification

The verification process is realized through the RAPTURE tool. To connect with RAPTURE, P_UPPAAL provides the RAPTURE tab. This tab is divided in two parts. At the top, you can write the initial and final state, and the probability to reach the final state from the initial state and, at the bottom, you can show the RAPTURE output.

To verify a property, the verifier has to translate first the system into textual form. This textual form will be input for RAPTURE.

In the figure 3.6, you can see the RAPTURE verifier running.

The next section treats the RAPTURE tool into detail.

3.7 RAPTURE

The context systems of RAPTURE are described in terms of Markov decision process [45], also called probabilistic transition systems (PTS) or probabilistic automata. This model allows to combine probabilistic and non-deterministic steps providing a natural extension to traditional non-deterministic models. The choice of this model is partly due to the fact that it is closed under parallel composition (which facilitates modeling and compositional reasoning), but primarily because PTSs are amenable to abstractions.

RAPTURE is focused on a restricted class of reachability properties. These properties allow to specify that the probability of reaching a particular condition ϕ_f from any reachable state satisfying a given initial condition ϕ_i is smaller (or greater) than a given probability p regardless of how non-deterministic choices of the model are resolved.

RAPTURE is based on automatic abstraction and refinement [19]. The basic idea is to use abstractions in order to reduce the high cost of the numerical analysis involved in computing the minimum and maximum reachability probabilities for PTSs. The abstractions considered are obtained via successive refinements, starting from an initial coarse partitioning of the state space derived from the property under study. For a given refinement the property is checked on the induced abstract model, hopefully settling the property. However, the verdict may be inconclusive, when threshold probability p happens to be between the calculated minimum and the maximum abstract probabilities. In this case, the abstraction is further refined and the property checked again. This process is successively repeated until either the property is settled, or no further refinement is possible. To efficiently store the state space, perform abstractions and process the refinement steps, RAPTURE use Bdds and Mtbdds (or Adds) [19, 20, 3].

3.7.1 Probabilistic Transitions Systems

As is pointed out in [19, 20], Probabilistic transition systems (PTS for short) generalize the well-known transition systems with probabilistic information. In a PTS, a transition does not lead to a single state but to a probability space whose sample space is a set of states. The model we define is widely used (see e.g. [30, 47]) and is also known as Markov decision processes [45].

Let $Distr(\Omega)$ denote the set of all probability distributions over the sample space Ω.

Definition 17. Probabilistic Transition Systems. *A probabilistic transition system (PTS for short) is a structure $T = (s, \rightarrow)$ where S is a set of states, and $\rightarrow \subseteq X \times Distr(S)$ is the transition relation. We write $s \rightarrow \pi$ for $(s, \pi) \in \rightarrow$. A PTS is said to be a fully probabilistic transition systems (FPTS for short) if $s \rightarrow \pi \wedge s \rightarrow \rho \Rightarrow \pi = \rho$. A rooted PTS (resp. FPTS) $(T, s_0)$is PTS (resp. FPTS) equipped with an initial state $s_0 \in S$. A PTS may be equipped with a proposition assignment $p : S \rightarrow \mathcal{P}(AP)$, where AP is a finite set of atoms and $\mathcal{P}(AP)$the set of propositional formula on AP. We define $\models \subseteq X \times \mathcal{P}(AP)$ by $s \models g$ iff $p(s) \Rightarrow g$ is a tautology.*

We write $s \rightarrow$ whenever there is a π such that $s \rightarrow \pi$; otherwise, we write $s \not\rightarrow$. We let $sup(\pi) = \{s' | \pi(s') > 0\}$. We call s a *sink* state if $s \not\rightarrow$.

Let $T = (S, \rightarrow)$ be a PTS. A *simple path* in T is a finite sequence of states $\sigma = s_0 s_1 s_2 \ldots s_n$, where for each $0 \leq i < n$ there exists $\pi_i \in Distr(S)$ such that $s_i \rightarrow \pi_i$ and $\pi_i(s_{i+1}) > 0$. Let $\sigma(i)$ denote the state in the i-th position. Let $|\sigma|$ be the length of σ and let $first(\sigma) = \sigma(1)$ and $last(\sigma) = \sigma(|\sigma|)$. A *simple path starting from* $s \in S$ is a simple path σ with $\sigma(1) = s$. A state t is *reachable* from another state s in T if there is a simple path in T with $s = first(\sigma)$ and $t = last(\sigma)$.

A *full path in* T is a sequence of states σ being either a simple path with $last(\sigma) \not\rightarrow$, or a infinite sequence. We denote by $s-paths(T)$ and $f-paths(T)$ the sets of simple paths and fullpaths in T starting from s. Let $reach(T, s)$ denote the set of all states reachable from s in T.

We now define a probability measure on the full paths of a FPTS F. For any simple path $\sigma \in s-paths(F)$, define $\sigma_\uparrow = \{\pi \in f-paths(F) | \sigma \leq \pi\}$ where $\leq$ is the classical prefix order on sequences. Let $\mathcal{F}(F)$ be the smallest σ-fied on $f - paths(F)$ which contains $\sigma_\uparrow$ for each $\sigma \in s - paths(F)$. Let $\mathcal{F}(F)$ such that for any $\sigma = s_0 s_1 \ldots s_n \in s - paths(F)$ such that for any $\sigma = s_0 s_1 \ldots s_n \in s - paths(F)$ such that $s_1 \rightarrow_F \pi_i$ for all $i, 0 \leq i < n$:

$$\mathrm{P}_{F,s}(\sigma_\uparrow) \triangleq \textbf{if } (s = s_0) \textbf{then} \pi_0(s_1) \cdot \pi_1(s_2) \cdot \ldots \cdot \pi_{n-1}(s_n) \textbf{else } 0$$

We will write $\mathrm{P}_{F,s}(\sigma)$ to denote $\mathrm{P}_{F,s}(\sigma_\uparrow)$. Intuitively, $\mathrm{P}_{F,s}(\sigma)$ is the probability of σ in F starting from s.

Any given PTS T defines a set of probabilistic executions, each one obtained by iteratively scheduling one of the possible post-state distributions from each pre-state, starting from a given state $s_0 \in S$. Notice that the same state s of T may occur more than once during a probabilistic execution and each time a different distribution from s may be scheduled. In order to distinguish such occurrences we include in all states s of a probabilistic execution the past history of s which is the unique path leading from the start state to s. Thus, a probabilistic execution essentially defines a finite or infinite ***tree***.

Definition 18. Probabilistic Execution *A probabilistic execution of a PTS* $T = (S, \rightarrow_T)$ *is a FPTS* $F = (s - paths(T), \rightarrow_F)$ *such that* $(q \rightarrow_F \rho) \Leftrightarrow (\exists \pi : last(q) \rightarrow_T \pi \wedge \forall s \in S : \rho(qs) = \pi(s))$

We denote by $execs(T, s_0)$ the set of all probabilistic execution of T rooted in s_0.

3.7.2 Computing Extremum Probabilities

For a given rooted PTS (T, s_0) we are interested in the extremum probabilities of reaching some final condition from a given initial condition. For any given formula $\phi \in \mathcal{P}(AP)$ we define the set of all minimal simple paths of T that end in a state satisfying condition ϕ as:

$$\Sigma_\phi^T \triangleq \{\sigma \in s - paths(T) | last(\sigma) \models_T \wedge \forall i, 0 < i < |\sigma| : \sigma(i) \models_T \neg\phi\}$$

By recording history information in states, the above set characterizes uniquely a set of simple paths of probabilistic executions of T. We also use Σ_ϕ^T to denote this alternative characterization. It should be clear from the context which alternative is used. We omit T in the notation whenever clear from context.

Definition 19. Extremum Probabilities *The minimun and maximum probabilities of reaching a final condition ϕ_f from an initial condition ϕ_i in a rooted PTS (T, s_0) equipped with a proposition assignment are defined respectively by*

$$\mathrm{P}^{inf}_{T,s_0}(\phi_i, \phi_f) \triangleq inf\{\mathrm{P}_{F,s_0}(\Sigma_{\phi f}) | s \in reach(T, S_0) \wedge s \models \phi_i \wedge (F, s) \in execs(T, s)\} \tag{3.1}$$

$$\mathrm{P}^{sup}_{T,s_0}(\phi_i, \phi_f) \triangleq sup\{\mathrm{P}_{F,s_0}(\Sigma_{\phi f}) | s \in reach(T, S_0) \wedge s \models \phi_i \wedge (F, s) \in execs(T, s)\} \tag{3.2}$$

We talk of an extremum probability to refer to either the infimum or supremum probability.

We use the shorthand $\mathrm{P}^{inf}(s)$ and $\mathrm{P}^{sup}(s)$ for $\mathrm{P}^{inf}_{T,s_0}(s = s_0, \phi_f)$ and $\mathrm{P}^{sup}_{T,s_0}(s = s_0, \phi_f)$ respectively. We denote by I and F the sets of states satisfying ϕ_i and ϕ_f, respectively. Our aim is to efficiently compute $\mathrm{P}^{inf}(I) \triangleq inf_{s \in I}\mathrm{P}^{inf}(s)$ and $\mathrm{P}^{sup}(F) \triangleq sup_{s \in I}\mathrm{P}^{sup}(s)$.

The equations 3.1 and 3.2 of definition 19define extremum probabilities, but do not provide an effective way of computing them. However, it is well known [4, 8] that P^{inf} and P^{sup} can be characterized as the least fixpoints of operators $\mathrm{F}^{inf}, \mathrm{F}^{sup} = (S \to [0, 1]) \to (S \to [0, 1]))$ defined as follows. If $s \in \mathrm{F}$ then $\mathrm{F}^{inf}(f)(s)\mathrm{F}^{sup}(f)(s) = 1$. If $s \notin \mathrm{F}$ then

$$\mathrm{F}^{inf}(f)(s) = \min_{s \to \pi} \sum_{s' \in S} \pi(s') \cdot f(s') \text{ and } \mathrm{F}^{sup}(f)(s) = \max_{s \to \pi} \sum_{s' \in S} \pi(s') \cdot f(s') \tag{3.3}$$

Based on the above equations, two methods have been explored to compute $\mathrm{P}^{inf}(s)$ and $\mathrm{P}^{sup}(s)$. One can either compute the least fixpoints by iterative methods, or the equations can be transformed into a linear optimization problem that can be solved using classical techniques of linear programming. RAPTURE implements the linear programming method.

We use a standard precomputation of certain sets of system states in order to simplify the system before applying linear programming techniques. These sets are: the set of all reachable states Reach, and for each $p \in \{0,1\}$ the set of states having infimum (resp. supremum) probability p of reaching ϕf. These latter sets of states are denoted $\mathrm{P}^{inf}_{=0}$, $\mathrm{P}^{sup}_{=0}$, $\mathrm{P}^{inf}_{=1}$, and $\mathrm{P}^{sup}_{=1}$ respectively. All of the above sets can be computed using discrete fixpoint analysis [22] on a boolean abstraction of the system.

Based on the above precomputations our linear programming problems for computing P^{inf} and P^{sup} become as follows:

maximize P^{inf} under the constraints

$$\begin{cases} \mathrm{P}^{inf} \leq \mathrm{P}^{inf}(s) & s \in I \\ \mathrm{P}^{inf}(s) = 0, & s \in \mathrm{P}^{sup}_{=0} \\ \mathrm{P}^{inf}(s) = 1, & s \in (F \cup \mathrm{P}^{inf}_{=1}) \\ \mathrm{P}^{inf}(s) = \sum_{s' \in S} \cdot \mathrm{P}^{sup}(s'), & s \to \pi, s \in S \backslash (\mathrm{P}^{sup}_{=0} \cup \mathrm{P}^{inf}_{=1} \cup F) \end{cases} \tag{3.4}$$

minimize P^{sup} under the constraints

$$\begin{cases} \mathrm{P}^{sup} \geq \mathrm{P}^{sup}(s) & s \in I \\ \mathrm{P}^{sup}(s) = 0, & s \in \mathrm{P}^{sup}_{=0} \\ \mathrm{P}^{sup}(s) = 1, & s \in (F \cup \mathrm{P}^{sup}_{=1}) \\ \mathrm{P}^{sup}(s) = \sum_{s' \in S} \cdot \mathrm{P}^{sup}(s'), & s \to \pi, s \in S \backslash (\mathrm{P}^{sup}_{=0} \cup \mathrm{P}^{sup}_{=1} \cup F) \end{cases} \tag{3.5}$$

3.7.3 Simulations and Partitioning

Probabilistic simulation [47, 29] is central to state the correctness of the abstraction technique showed here. For any $\delta \in Distr(S \times S), s \in S$ and $X \subseteq S, \delta(s, X)$ and $\delta(X, s)$ will denote resp. $\Sigma_{x \in X}\delta(s, x)$ and $\Sigma_{x \in X}\delta(x, s)$.

Definition 20. Simulation *Let $C \subseteq S \times S$ be a relation on states defining a discrimination criterion. A relation $R \subseteq S \times S$ is a C-(probabilistic) simulation if, whenever sRt,*

1. *$(s, t) \in C$ and*

2. *if $s \to \pi$, there exist ρ such that $t \to \rho$ and $\pi \sqsubseteq_R \rho$.*

where $\pi \sqsubseteq_R \rho$ if there is $\delta \in Distr(S \times S)$ such that for all $s, t \in S$, $(i)\pi(s) = \delta(s, S)$, $(ii)\rho(t) = \delta(S, t)$ and $(iii)\delta(s, t) > 0 \Rightarrow sRt$. s is C-simulated by t, notation $s \preceq_C t$, there is a C-simulation R with sRt.

Our interest is to check when a PTS reaches a goal ϕ_f starting from any state satisfying some initial condition $\phi_i (\phi_i, \phi_f \in \mathcal{P}(AP))$. Let C_phi_i, ϕ_f be the discriminating criterion defined by

$$(s, t) \in C_{\phi_i, \phi_f} \Leftrightarrow (s \models \phi_f \Leftrightarrow t \models \phi_f) \wedge (s \models \phi_i \Leftrightarrow t \models \phi_i)$$

We write only C whenever ϕ_i and ϕ_f are clear form the context. Notice that C is equivalence relation. Simulation $\preceq_C$ provides a sufficient condition for preservation of extremum probabilities, as made precise by the following theorem.

Theorem 1. *Let (T_1, s_0^1) and (T_2, s_0^2) be two rooted PTS such that none of them contains a sink state, and let $C = C_{\phi_i, \phi_f}$. Then $(T_1, s_0^1) \preceq_C (T_2, s_0^2)$ implies $\mathrm{P}^{sup}_{T_1, s_0^1}(\phi_i, \phi_f) \leq \mathrm{P}^{sup}_{T_2, s_0^2}(\phi_i, \phi_f)$ and $\mathrm{P}^{inf}_{T_1, s_0^1}(\phi_i, \phi_f) \geq \mathrm{P}^{inf}_{T_2, s_0^2}(\phi_i, \phi_f)$*

The requirement that every state has a transition is not really harmful as each sink state can always be completed with a self-looping transition without affecting the properties of our interest on the original PTS. We can abstract a PTS by partitioning its state space, and any such partitioning will induce an abstract PTS which will simulate the original (concrete) one. As a result extremum properties will be preserved by the abstract system.

Definition 21. Quotient PTS *Let $T = (S, \rightarrow_T)$ a PTS equipped with the proposition assignment p. Let $\mathcal{A} = (A_k)_{k \in K}$ be a partition of S. The quotient PTS according to $\mathcal{A}$ is the PTS $T/\mathcal{A} = (\mathcal{A}, \rightarrow_{\mathcal{A}}, p/\mathcal{A})$, where*

1. $A \rightarrow_{\mathcal{A}} \pi/\mathcal{A} \Leftrightarrow \exists s \in A : s \rightarrow \pi \wedge \forall A' \in \mathcal{A} : (\pi/\mathcal{A})(A') \triangleq \Sigma_{s' \in A} \pi(s')$ *and*

2. $p/\mathcal{A}(A) \triangleq \bigvee_{s \in A} p(s)$

For a rooted PTS (T, s_0), its quotient is given by $(T, s_0)/\mathcal{A}, A$ provided $s_0 \in A \in \mathcal{A}$.

Theorem 2. *Let (T, s_0) be a rooted PTS with a set of states S and let C be an equivalence relation defining a partition $\mathcal{A}$ of S. Then for any partition $\mathcal{B}$ of S such that $\mathcal{B} \leq \mathcal{A}$, $(T, s_0)/\mathcal{B} \preceq_C (T, S_0)/\mathcal{A}$.*

Notice the special case of the theorem where $\mathcal{B}$ partitions S into singleton sets. In this case $(T, s_0)/\mathcal{B}$ is isomorphic to (T, s_0). The following corollary states the relationship of abstraction by partitioning and preservation of extremum probabilities.

Corollary 1. *Let (T, s_0) be a rooted PTS equipped with a propositon assignment. Let ϕ_i and ϕ_f be the initial and final conditions. Let C be the equivalence relation $C = C_{\phi_i, \phi_f}$ defining a partition $\mathcal{C}$ of S. Then for any two partitions $\mathcal{A}$ and $\mathcal{B}$ such that $\mathcal{B} \leq \mathcal{A} \leq \mathcal{C}$,*

$$\mathrm{P}^{sup}_{(T,s_0)/\mathcal{B}}(\phi_i, \phi_f) \leq \mathrm{P}^{sup}_{(T,s_0)/\mathcal{A}}(\phi_i, \phi_f) \text{ and } \mathrm{P}^{inf}_{(T,s_0)/\mathcal{B}}(\phi_i, \phi_f) \geq \mathrm{P}^{inf}_{(T,s_0)/\mathcal{A}}(\phi_i, \phi_f)$$

3.7.4 RAPTURE Architecture Implementation

It's architecture (see Figure 3.7) is the following:

1. *The frontend* parses the input language, that specifies both the system to be analyzed, the property and possibly the components (processes and variables) not abstracted in the inital abstraction. The output is a symbolic representation of the system (i.e., the probabilistic transition function and sets of root, initial and final states).

2. *Boolean analysis* is then performed. It if allows to prove or disprove the property, the verdict is emitted.

3. Otherwise, *the initial abstraction* is build, and the verification process alternating numerical analysis and refinement steps starts.

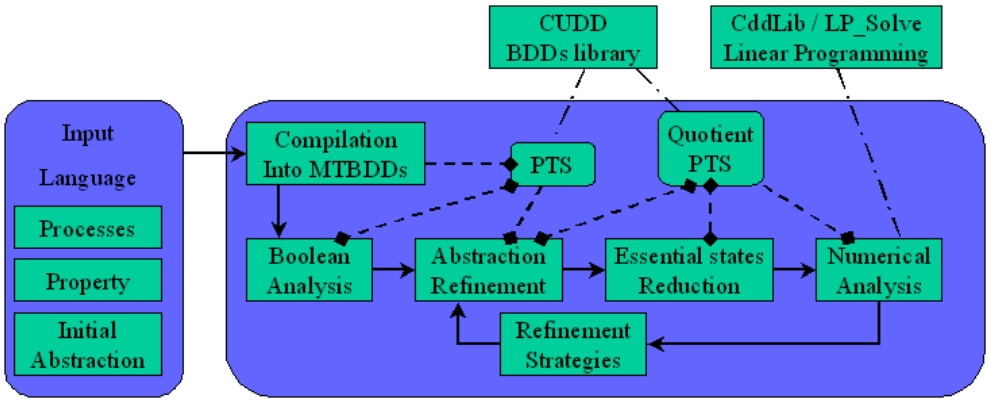

Figure 3.7: Architecture of the RAPTURE tool

As stated before, RAPTURE uses linear programing to compute extremum probabilities. Because of precision problems arising with some case studies, RAPTURE offers the following possibilities:

- Use of the sparse matrix based solver LP_SOLVE,with coefficients being ordinary real numbers (1);
- Use of the dense matrix based solver CDDLIB, with coefficients being either exact rational numbers (2), multi-precision floating point numbers (3), or ordinary real numbers.

The possibilities that give the best results are (1) and (2). (1) is better whenever there is no precision problems, because it uses sparse matrices and our LP problems are sparse, whereas (2) is very useful when precision problems arise and/or when exact results are wanted.

3.8 An example

Communication through unreliable media

Let us describe a small example. We will model a client and a server which communicate through two unreliable channels (a channel for each way). The client has to send a sequence of M different requests to the server. The server treats the request and send an acknowledgement. If there is a failure (we suppose that the

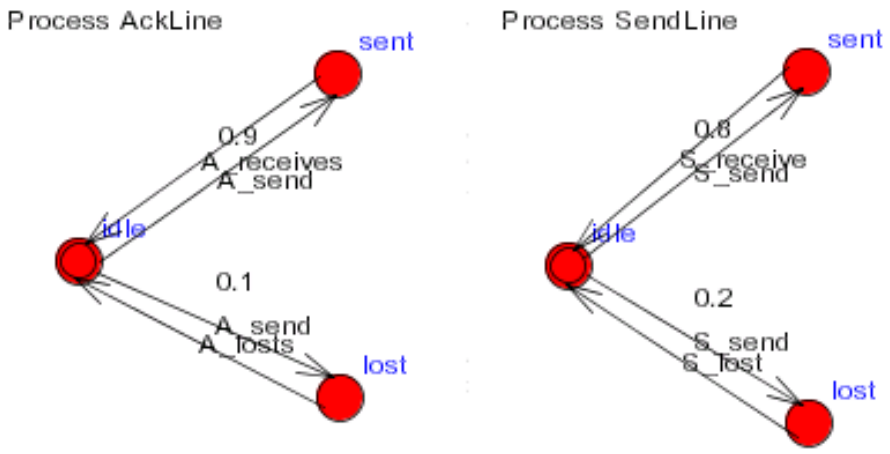

Figure 3.8: The two communication media: AckLine & SendLine

client can detect it), the client can retry twice to send the request. On completion, the sender emit the message CORR, otherwise the message ERR will be emitted.

Global Declarations

We first declare the set of channels:

channels: S_send,S_receive,S_lost,A_receives,A_receivecA_losts,A_lostc, START,CORR,ERR;

The two communication media

We define the SendLine and AckLine processes(see figure 3.8). The field synchronizations allows us to define the set of channels on which the process synchronize.

The Client process

The more complex process is the Client process that you can see in the figure 3.9. This process owns two local variables. Their type is uint(2), which means: 2-bits unsigned integers.

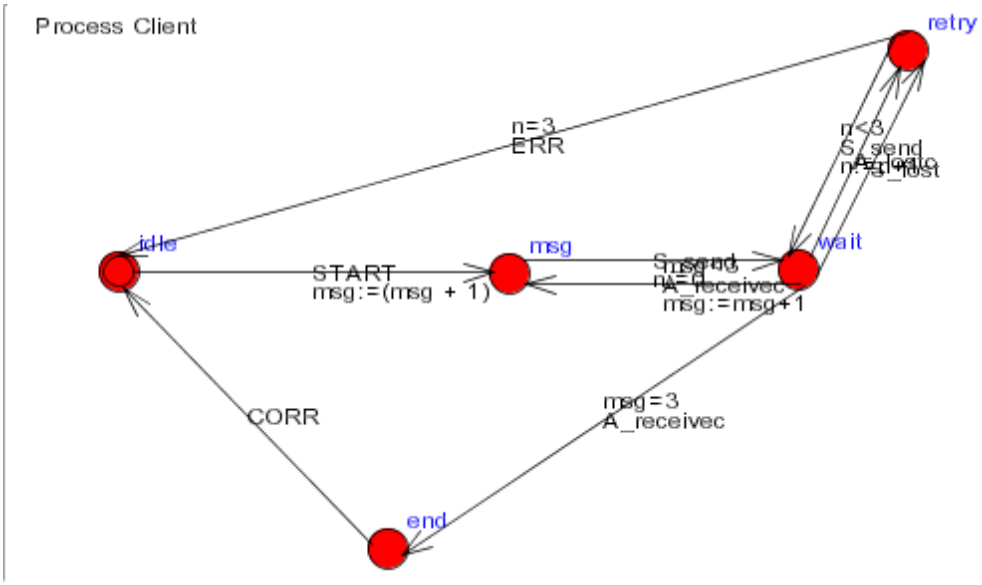

Figure 3.9: The client process

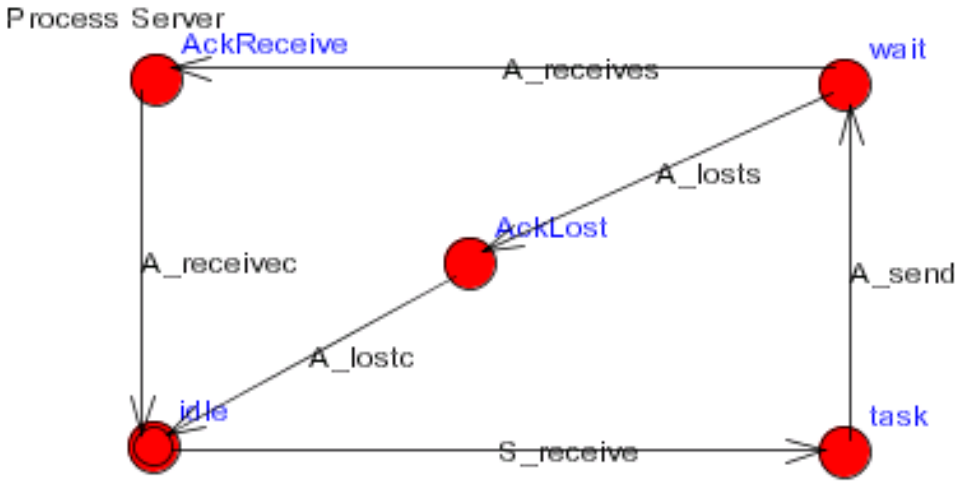

Figure 3.10: The server process

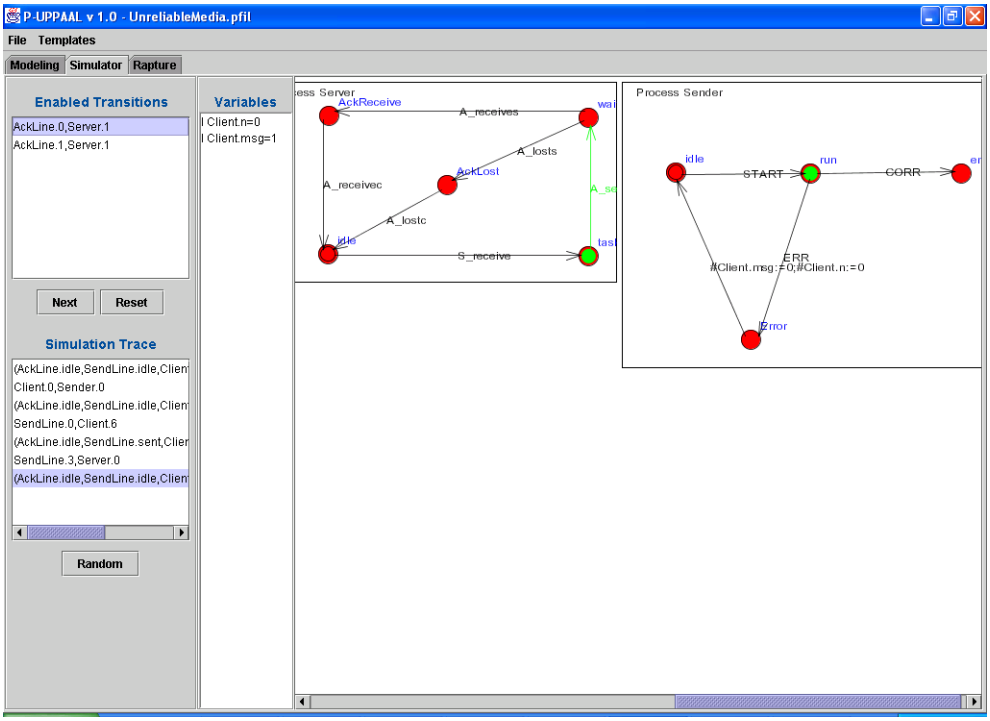

Figure 3.11: The system simulation

The Server process

Last, we have the (dummy) Server process that is shown in the figure 3.10.

The Simulation see figure 3.11

The property and the verification

To check is the probability to send correctly one message. Let's add the process sender into the system (see the figure 3.12) .

To specify the property we set initial to "#Sender.idle" and final to "#Sender.end"

The output provides by RAPTURE can be seen in the figures ?? and ??

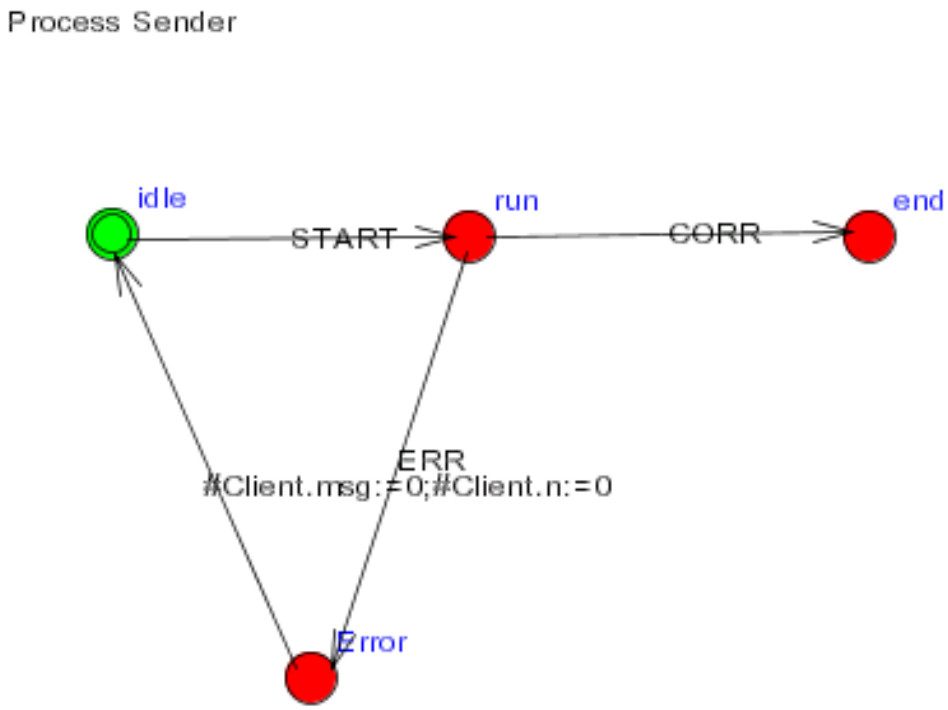

Figure 3.12: The sender process

The property is false, because the $\mathrm{P}^{sup} \leq 0.5$ on the stabilized partition of the system. As every member of this partition are bisimulation equivalence classes, we get that $\mathrm{P}^{sup} = 0.975639513753$ on the concrete system.

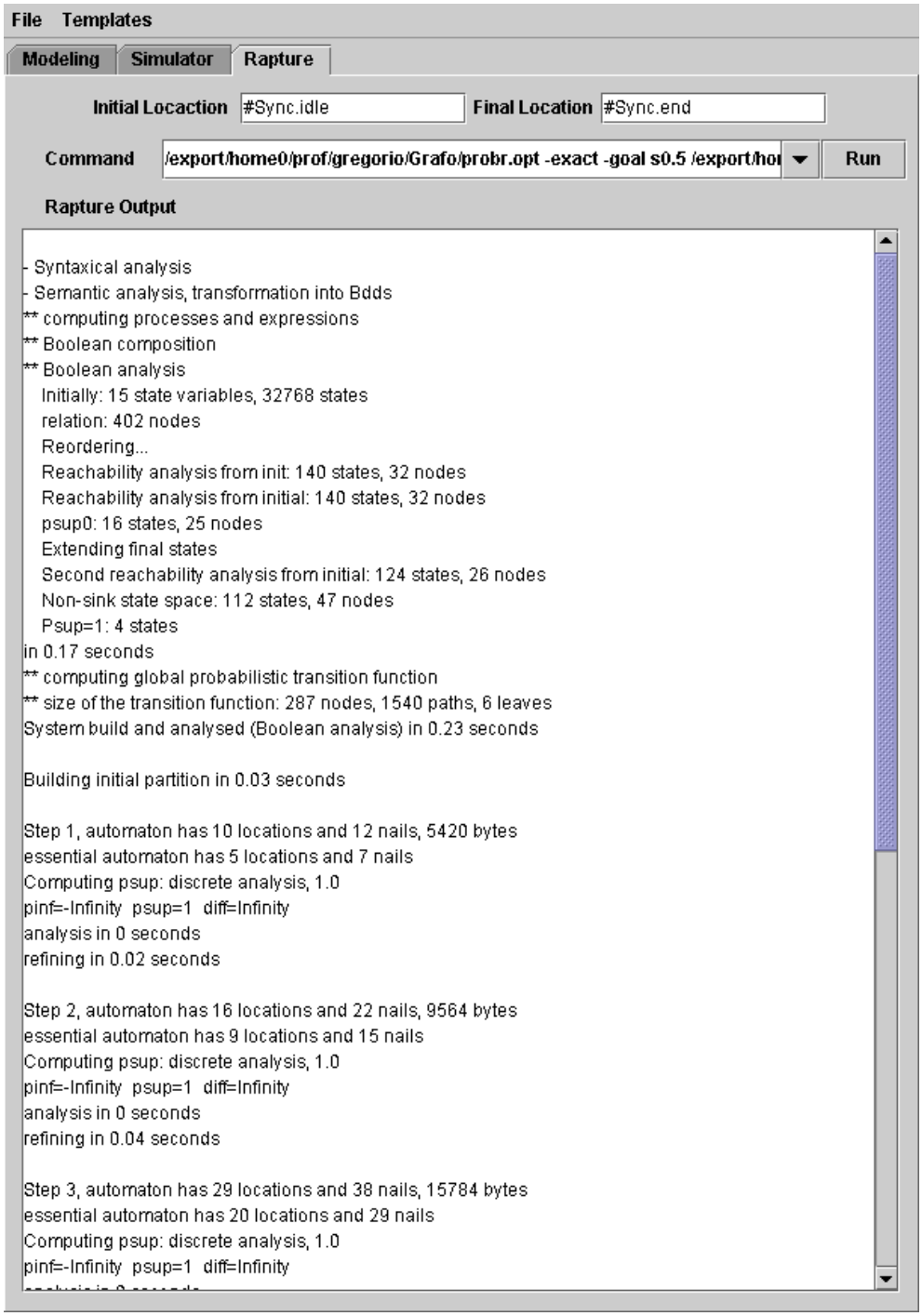

Figure 3.13: The verifier verdict

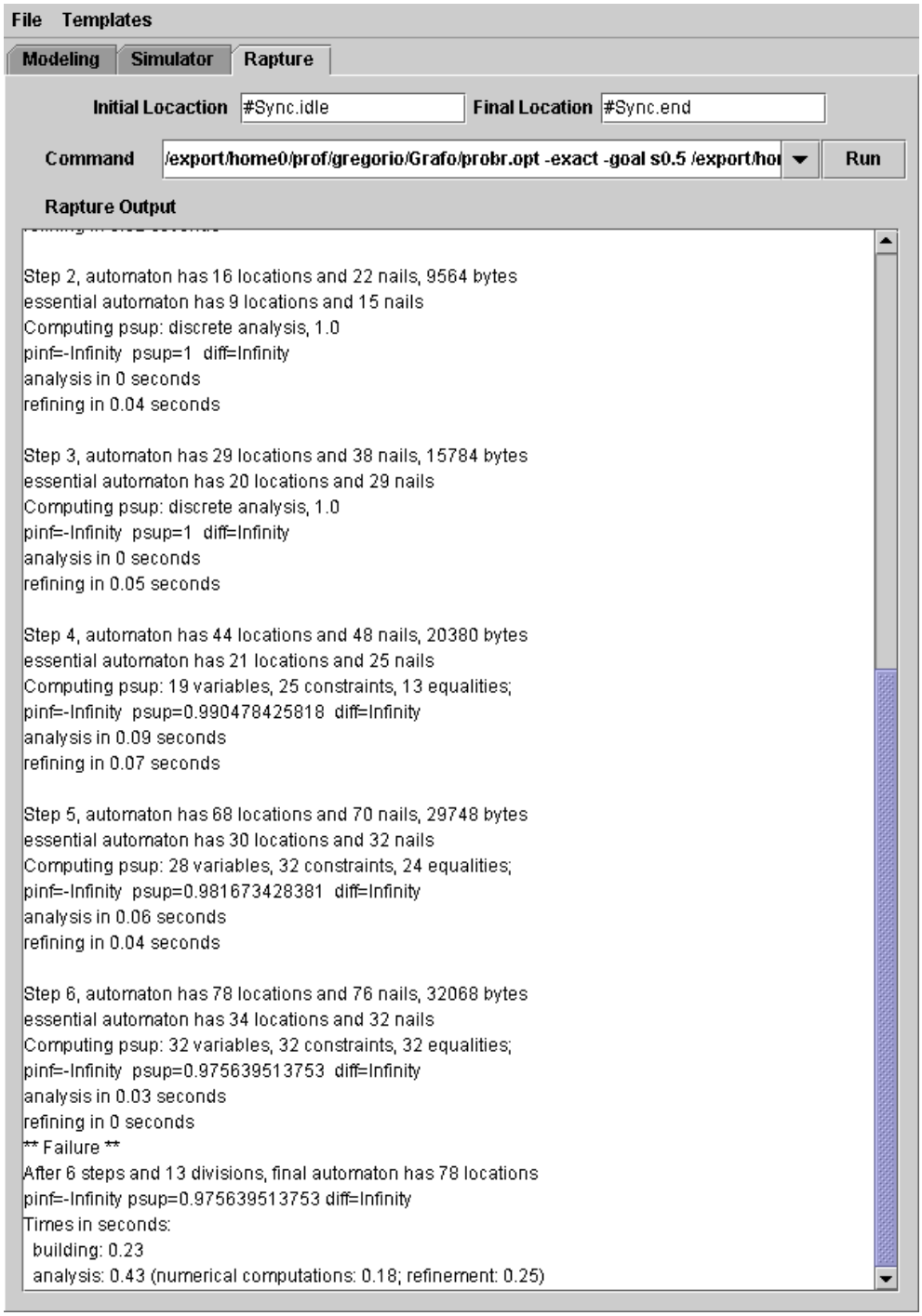

Figure 3.14: The verifier verdict

Chapter 4

Conclusions & Future Research

4.1 Conclusions

In this work, we have shown two ways for studying system correctness. These ways are different from traditional bisimulation and testing. Both of them are based in the same idea, i.e, *"Verify the system"*. One of them treats the temporal behaviour and the other one treats the probabilistic behaviour, i.e, we refer to the tools UPPAAL and P_UPPAAL respectively.

Thus, the ***UPPAAL*** tool treats system temporal behaviour. To make it, UPPAAL models are represented as networks of timed automata that allows us to introduce the time through the variables named clocks. Furthermore, the timed automata provide us the invariants and the guards that implements temporal requirements.

On the other hand, ***P_UPPAAL*** tool treats system probabilistic behaviour. As UPPAAL uses timed automata, P_UPPAAL uses probabilistic transitions systems (PTS) to represent the model. And while UPPAAL uses invariants and guards, P_UPPAAL adds a new transition field called probability that represents the probability to make this transition.

Both of them, UPPAAL and P_UPPAAL provide a way to validate the system. In both, you can simulate the systems under study that allows us to prove system behaviour.

Finally, both UPPAAL and P_UPPAAL can verify the system. UPPAAL uses a symbolic model checking based on reachability analysis by constraint solving that verify safety properties for example. However, P_UPPAAL, through RAPTURE, uses a reachability analysis based on probabilistic simulations over an abstraction of the system that reduces the cost of the numerical analysis. This analysis calculates the extremum probabilities (minimum and maximum) to reach a final condition from a reachable state that holds a initial condition.

In short, P_UPPAAL shows a different way to automatic verification for other kind of properties.

Other main characteristic of P_UPPAAL is that P_UPPAAL is able to import UPPAAL models. It is a important goal in this work, because it allows us to capture the probabilistic behaviour of Real Time Systems. It means that P_UPPAAL can connect with UPPAAL to expand UPPAAL features.

Other important point for this work is that this work has been developed in cooperation with Kim Guldstrand Larsen Professor in the Department of Computer Science at Aalborg University within the Distributed Systems and Semantics Unit, and part-time industrial professor at the Formal Methods and Tools Group, Twente University. Also, he is the director of the Aalborg section of Basic Research in Computer Science, BRICS , and co-director of the newly started Center for Embedded Software Systems, CISS. I have been staying at Aalborg University, Department of Computer Science for the period of 6 month in spring 2002, where I have been researching as a Ph. D. student. I have been interacting with others colleagues at the Department during my stay and Kim G. Larsen was my tutor. I have also been taking part in their activities in developing and applying the real-time validation tool UPPAAL.

4.2 Future Research

Although, P_UPPAAL is in a high degree of development, there are a several points that have to be discussed into detail. For example, P_UPPAAL models can be exported to UPPAAL models. This idea is similar to the developed import process. Thus, we might use

the same principle: "PTS and Timed Automate are derived from labeled transition systems". Furthermore, if we extend this idea, we may generate a label transition system that may be exported to all its derived classes.

Thus, because of the high degree of development, it is in time to prove the system. To prove the system, we must introduce use cases. We are interested in implementing the video standard MPEG-2 and Ghost Packet algorithm.

The reason for choosing MPEG-2 is due the fact that this protocol has been studied for time at my research group, as pointed out in [50, 48, 44]. The basis of MPEG-2 is that it addresses the combining of one or more elementary streams of video and audio, as well as, other data into single or multiple streams which are suitable for storage or transmission.

Ghost Packet algorithm, has been chosen becouse this algorithm was my master thesis goal. This work was accepted at "The international conference on parallel processing 2001" IEEE proceedings under the name "Performance Issues of Deterministic and Adaptive Ghost-Packet Routers" [12]. In this paper, we presented a new starvation and deadlock free routing algorithm.

Much work is still left, but an other step has been made towards capturing system behaviours and its verification.

Bibliography

[1] Luca Aceto, Augusto Bergueno, and Kim G. Larsen., *Model checking via reachability testing for timed automata*, In Proceedings of the 4th International Workshop on Tools and Algorithms for the Construction and Analysis of Systems. (31 March - 2 April, 1998), 263–280.

[2] R. Alur and D. Dill, *Automata for modeling real–time systems*, Springer–Verlag, 1990.

[3] R.I. Bahar, E.A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, *Algebraic decision diagrams and their applications*, ICCAD-93: ACM/IEEE International Conference on Computer Aided Design.

[4] Christel Baier, *On algorithmic verification methods for probabilistic systems*, Ph.D. thesis, Faculty for Mathematics and Informatics, University of Mannheim, 1998.

[5] Gerd Behrmann, Thomas Hune, and Frits Vaandrager, *Distributed timed model checking - How the search order matters*, Proc. of 12th International Conference on Computer Aided Verification (Chicago), Lecture Notes in Computer Science, Springer-Verlag, Juli 2000.

[6] Johan Bengtsson, W.O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi, *Verification of an Audio Protocol with Bus Collision Using* Uppaal, no. 1102, Springer–Verlag, July 1996, pp. 244–256.

[7] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, Yi Wang, and Carsten Weise, *New Generation of* Uppaal, Int. Workshop on Software Tools for Technology Transfer, June 1998.

[8] A. Bianco and L. de Alfaro, *Model checking of probabilistic and nondeterministic and non-deterministic systems*, LNCS, no. 1026, 1995.

[9] H. Bowman, G. Faconti, J-P. Katoen, D. Latella, and M. Massink, *Automatic Verification of a Lip Synchronisation Algorithm using UPPAAL*, In Proceedings of the 3rd International Workshop on Formal Methods for Industrial Critical Systems (Amsterdam, The Netherlands), Jan Friso Groote and Bas Luttik and Jos van Wamel, 1998.

[10] H. Bowman, G. Faconti, and M. Massink, *Specification and verification of media constraints using UPPAAL*, In 5th Eurographics Workshop on the Design, Specification and Verification of Interactive Systems, Eurographics Series, no. 1384, Springer–Verlag, August 1998, pp. 281–297.

[11] A. Bueno, V. Valero, and F. Cuartero, *TPAL: A timed-probabilistic model for concurrent processes*, Proceedings of APSEC'97/ICSC'97 (Hong-Kong), 1997.

[12] M. C. Carrión, G. Díaz, and B. Caminero, *Performance issues of deterministic and adaptive ghost-packet routers*, Proceeding of the 2001 International Conference on Parallel Processing (Valencia, Spain), IEEE Computer Society, 3–7 September 2001, pp. 33–40.

[13] D. Cazorla, *PNAL: Un modelo algebraico para procesos probabilísticos y no deterministas*, Ph.D. thesis, Dep. of Computer Science. University of Castilla-La Mancha, 2001.

[14] D. Cazorla, F. Cuartero, V. Valero, and F.L. Pelayo, *Reasoning about probabilistic and nondeterministic processes*, Actas de las I Jornadas sobre Programación y Lenguajes, PROLE 2001 (Almagro, Ciudad Real (Spain)), 2001.

[15] Diego Cazorla, Fernando Cuartero, Valentín Valero, Fernando L. Pelayo, and J. José Pardo, *Algebraic theory of probabilistic and nondeterministic processes*, The Journal of Logic and Algebraic Programming (2002), To appear.

[16] E. M. Clarke and E. A. Emerson, *Automatic verification of finite-state concurrent systems using temporal logic specifications.*, In proceedings of the 10th Annual ACM Symposium on Principles of Programming Language, january 1983.

[17] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled., *Model checking*, MIT Press, 1999.

[18] E.M. Clarke and H. Schlingloff, *Model checking*, A. Robinson and A. Vornkov, 2000.

[19] Pedro R. D'Argenio, BertrandbJeannet, Henrik E. Jensen, and Kim G. Larsen, *Reachability analysis of probabilistic systems by successive refinements*, PAPM-PROBM 2001 (Aachen (Germany)), LNCS, vol. 2165, September 2001.

[20] Pedro R. D'Argenio, Bertrand Jeannet, Henrik E. Jensen, and Kim G. Larsen, *Reduction and refinement strategies for probabilistic analysis*, Process Algebra and Probabilistic Methods - Performance Modelling and Verification, PAPM-PROBMIV 2002 (Copenhagen (Denmark)), LNCS, vol. 2399, July 2002.

[21] P.R. D'Argenio, J.-P. Katoen, T.C. Ruys, and J. Tretmans., *The bounded retransmission protocol must be on time!*, In Proceedings of the 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems. (1997), 416–431.

[22] Luca de Alfaro, *Formal verification of probabilistic systems*, Phd thesis, Stanford University, 1997.

[23] Nicolas Halbwachs, *Delay analysis in synchronous programs*, Lecture Notes in Computer Science, no. 697, Springer-Verlag, 1993.

[24] Klaus Havelund, Kim G. Larsen, and Arne Skou, *Formal verification of a power controller using the real–time model checker uppaal*, 5th International AMAST Workshop on Real–Time and Probabilistic Systems (1999).

[25] Klaus Havelund, Arne Skou, Kim G. Larsen, and Kristian Lund., *Formal modelling and analysis of an audio/video protocol: An industrial case study using uppaal*, (3–5 December 1997), 2–13.

[26] H. Hüttel, K. G. Larsen, and C. Weise, *The use of static sontructs in a modal process logic*, Lecture Notes in Computer Science, no. 363, Springer-Verlag, 1989.

[27] Bertrand Jeannet, Pedro R. D'Argenio, and Kim G. Larsen, *RAPTURE: A tool for verifying markov decision processes*, Tools Day, International Conference on Concurrency Theory, CONCUR'02 (Brno (Czech Republic)), August 2002, Technical Report, Faculty of Informatics at Masaryk University Brno.

[28] Henrik Ejersbo Jensen, Kim G. Larsen, and Arne Skou., *Modelling and analysis of a collision avoidance protocol using spin and uppaal.*, In Proceedings of the 2nd SPIN Workshop. (5 August 1996.).

[29] B. Jonsson and K. Larsen, *Specification and refinement of probabilistic processes*, July 1991.

[30] Bengt Jonsson, Kim G. Larsen, and Wang Yi, *Probabilistic process algebra*, E-HPA, 2001, pp. 685–710.

[31] Pardo Juan Jose, Valero Valentín, Ruiz Mª Carmen, and Cambronero Mª Emilia, *New features of tpal for the specification and analisis of concurrent systems*, Tech. Report DIAB-02-01-25, UCLM, 2002, http://www.info-ab.uclm.es/sec-ab/Tecrep/diab020125.ps.zip.

[32] Joost-Pieter Katoen, *Concepts, algorithms, and tools for model checking*, 1998/1999, Lecture Notes of the course Mechanized Validation of Parallel Systems (course number 10359).

[33] François Laroussinie and Kim G. Larsen, *Compositional model checking of real time systems*, LNCS-962, 1995, RS-95-19, pp. 27–41.

[34] Kim G. Larsen, Paul Pettersson, and Wang Yi, *Compositional and Symbolic Model-Checking of Real-Time Systems*, Proc. of the 16*th* IEEE Real-Time Systems Symposium, IEEE Computer Society Press, Dec 1995, pp. 76–87.

[35] Kim G. Larsen, Paul Pettersson, and Wang Yi, *Diagnostic Model–Checking for Real–Time Systems*, Proc. of Workshop on Verification and Control of Hybrid Systems III, no. 1066, Springer–Verlag, October 1995, pp. 575–586.

[36] Kim G. Larsen, Paul Pettersson, and Wang Yi, UPPAAL *in a Nutshell*, Int. Journal on Software Tools for Technology Transfer **1** (1997), no. 1–2, 134–152.

[37] Kim G. Larsen, Paul Pettersson, and Wang Yi, UPPAAL*: Status and developments*, no. 1254, Springer–Verlag, June 1997, pp. 456–459.

[38] Fredrik Larsson, Kim G. Larsen, Paul Pettersson, and Wang Yi, *Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction*, Proc. of the *18*th *IEEE Real-Time Systems Symposium*, IEEE Computer Society Press, December 1997, pp. 14–24.

[39] Magnus Lindahl, Paul Pettersson, and Wang Yi, *Formal Design and Analysis of a Gear-Box Controller*, Proc. of the 4*th* Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, no. 1384, Springer–Verlag, March 1998, pp. 281–297.

[40] Henrik Lönn and Paul Pettersson, *Formal Verification of a TDMA Protocol Startup Mechanism*, Proc. of the *Pacific Rim Int. Symp. on Fault-Tolerant Systems*, December 1997, pp. 235–242.

[41] A. Olivero, J. Sifakis, and S. Yovine, *Using abstractions for the verification of linear hybrid systems*, Lecture Notes in Computer Science, no. 818, Springer-Verlag, July 1994, pp. 81–94.

[42] J.J. Pardo, V. Valero, F. Cuartero, and D. Cazorla, *Automatic translation of a timed process algebra into dynamic state graphs.*, Proceeding of Asian Pacific Software Engineering Conference, IEEE Computer Society, December 2001, pp. 63–70.

[43] J.J. Pardo, V. Valero, and M.Carmen Ruiz, *Tpal: Una herramienta de simulación de sistemas concurrentes con restricciones temporales*, Sistemas Distribuidos: Modelos y Aplicaciones, Albacete(Spain), July 2001, pp. 157–169.

[44] F.L. Pelayo, F. Cuartero, V. Valero, D. Cazorla, and T. Olivares, *Specification and Performance of the MPEG-2 Video Encoder by Using the Stochastic Process Algebra: Rosa*, Proceedings of the 17th Annual UK Performance Engineering Workshop. UKPEW'2001. (Mourad Kara K. Djemame, ed.), 2001, pp. 105–116.

[45] M.L. Puterman, *Markov decision processes: Discrete stochastic dynamic programming*, Willey series in probability and mathematical statistics, Jhon Wiley and Sons, 1994.

[46] A.W. Roscoe, *The theory and practice of concurrency*, Prentice Hall, 1998.

[47] R. Segala, *Modeling and verification of randomized distributed real-time systems*, Phd thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, June 1995.

[48] Valentín Valero, Fernando L.Pelayo, Fernando Cuartero, and Diego Cazorla, *Specification and Analysis of the MPEG-2 Encoder with Timed-Arc Petri Nets*, Electronic Notes in Theoretical Computer Science **66** (2002), no. 2.

[49] Valentín Valero, J.J. Pardo, and Fernando Cuartero, *Translating TPAL Specifications into Timed-Arc Petri Nets*, Proceedings of ICTAPN'2002 (Adelaine, Australia), June 2002.

[50] Valentín Valero, Fernando L. Pelayo, Fernando Cuartero, and Diego Cazorla, *Analysis of the MPEG-2 Encoder Algorithm with Timed-Arc Petri Nets*, Actas de las X Jornadas de Concurrencia (Jaca, Spain), Editorial Kronos, June 2002, pp. 71–86.

[51] Wang Yi, Paul Pettersson, and Mats Daniels, *Automatic Verification of Real-Time Communicating Systems By Constraint-Solving*, Proc. of the 7*th* Int. Conf. on Formal Description Techniques (Dieter Hogrefe and Stefan Leue, eds.), North–Holland, 1994, pp. 223–238.