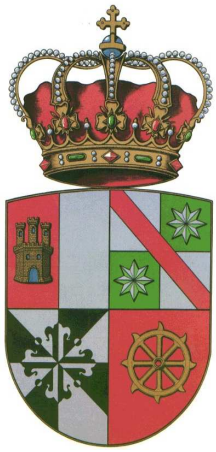
Applying and Developing Formal Techniques in the Design of E-Commerce Oriented Web Services with Strong Time Constraints



Dissertation for the degree of Doctor of Computer Science to be presented with due permission of the Department of Computer Science, for public examination and debate.

University of Castilla-La Mancha

Author: Gregorio Díaz Descalzo

Supervisors: Dr. Fernando Cuartero Gómez

Dr. Valentín Valero Ruiz

To the memory of my sister,
Maria del Carmen, who devoted her life
to the research in the university.

PREFACE

This dissertation is the succeed of the study and research at the group of Concurrency and Formal Methods of the University of Castilla – La Mancha from 2001 to 2005. I am grateful to my tutors Dr. Fernando Cuartero Gómez and Dr. Valentín Valero Ruiz for theirs support and also to my group colleagues for their professional and personal support.

The work was funded by the Escuela Politécnica Superior de Albacete at the Department of Computer Science, currently called Department of Computing System, and the Distributed Systems and Semantics group at Department of Computer Science in Aalborg University, the CICYT and JCCM projects and by personal grants from La Junta de Comunidades de Castilla – La Mancha. Without the founding and grants from these entities, the full-time research and international visits would not have been possible.

I would like to thank my wife Esther and my parents Gregorio and Juana Maria for their support and encouragement.

In the end, but not, in my mind, I would like to thank to my pals in the doctoral studies, whose personal support cannot be measured in words. How many hours have we spent in the university by listening strange songs? How many complaints and god moments have we shared?

For all of this, to everyone whit whom I have shared this path, thanks.

Contents

List of Figures

CHAPTER 1

Introduction

Our society is experimenting a change due to the communication systems, which have grown in importance since the development of new techniques to make them faster and better. Examples as the development of the telegraph in the 19th century show how fast the communication world has changed if we compare it with other technologies.

The most recent communication system is Internet. Internet has provided to our society with powerful tools like the World Wide Web, or WWW for short, which has granted us the access to the biggest library never created. But it is not optimized at all because the information indexed by Internet is not always useful. However, the added value by Internet to an information based world makes it a lesser drawback.

Thus, Internet has provided us with a capability to communicate our software or hardware systems. This capability allows us to synchronize these systems in order to communicate one another and the synchronization between them allows the task distribution and specialization. These effects are similar to the effects derived from the development of our main communication system, the language. For example, a consequence of the language capability together with a sedentary lifestyle in widely communities was that task distribution was done in groups and these groups specialized themselves in these tasks. This scenario is visible for example in prehistoric life when men specialized themselves

in tasks as haunting, women and children specialized themselves in tasks as harvesting.

Currently, Internet has achieved a high level of development as communication system for business processes. For example, the security issues in a payment process are one of the most important goals that must be covered by current applications on Internet and, in order to accomplish this goal, protocols such as TLS, SSL, SET, etc. have been developed. Furthermore, due to the heterogeneous nature of Internet, standardization issues have been covered by some entities, like w3c, OMG, OASIS, etc. Now, one of the most important challenges is the development of business processes by using Web Services. Thus, as in any other development process, it is important to guarantee that the system behaves as expected, and therefore, an important goal is the validation and Verification of Web Services.

1.1 Motivation

As shown before, the society model is changing. Our society is based on the information exchange due to the growth of Internet and Telecommunications. On the European Union the annual expenditure on ICT (Information and Communication Technology) amounted to an estimated of more than 500 billion EUROS, which was approximately the 6% of the total Gross Domestic Product. And the Internet access has increased for households and enterprises. In 2003, the access level of household to Internet was about 45%. The access of enterprisers was higher, reaching in some countries over 90% of all enterprises (source: EUROSTAT [139]).

Due to this change in the society model it becomes necessary to increase the research on the development of systems based in Internet, whose objective is to develop solutions for automating their peer-to-peer collaborations, in an effort to improve productivity and reduce operating costs.

In the last years some new techniques and languages for developing distributed application have appeared, such as the Extensible Markup Language, XML, and some new Web Services frameworks [34, 64, 137] for describing interoperable data and platform neutral business interfaces, enabling more open

business transactions to be developed.

Web Services are a key component of the emerging, loosely coupled, Web-based computing architecture. A Web Service is an autonomous, standards-based component whose public interfaces are defined and described using XML [85]. Other systems may interact with a Web Service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols.

The Web Services specifications offer a communication bridge between the heterogeneous computational environments used to develop and host applications. The future of E-Business applications requires the ability to perform long-lived, peer-to-peer collaborations between the participating services, within or across the trusted domains of an organization.

Web Services cover a wide range of systems, which in many cases have strong time constraints (for instance, peer-to-peer collaborations may have time limits to be completed). Then, in many Web Services descriptions these time aspects can become very important. Actually, they are currently covered by the top level layers in Web Services architectures with elements such as time-outs and alignments. Time-outs allow each party to fix the available time for an action to occur, while alignments are synchronizations between two peer-to-peer parties.

Thus, it becomes important for Web Services frameworks to ensure the correctness of systems with time constraints. For instance, we can think in a failure of a bank to receive a large electronic funds transfer on time, which may result in huge financial losses. Then, there is growing consensus that the use of formal methods, development methods based on some formalism, could have significant benefits in developing E-business systems due to the enhanced rigor these methods bring [70]. Furthermore, these formalisms allow us to reason with the constructed models, analyzing and verifying some properties of interest of the described systems. One of these formalisms are timed automata [2], which are very used in model checking [32], and there are some well-known tools supporting them, like UPPAAL [52, 53, 93] and KRONOS [19].

1.2 Objectives

The experience that software engineering has achieved with the pass of time has shown that errors, when the software is in use, have a high cost. It usually occurs because these errors came from the first phases in software life cycle. It means that the software engineer must back again to these phases to solve the problem. It implies that the engineer must repeat every phase of software development and it also implies that the software cost is increased at least by double. Some techniques have been developed to solve these problems and usually are based in formal techniques that ensure the system correctness. These techniques are based on formal languages, which allow us to describe the behavior of systems in a precise and unambiguous way, as well as to apply some existing techniques to ensure the correctness. By the use of these techniques we can get an abstract model of the system that allows software engineers to check the system before its implementation.

The main objective in this dissertation is the application and development of formal techniques to avoid these kinds of errors. More specifically, we intend to use formal methods to develop some techniques to guarantee the correctness of distributed applications and we will emphasize their use on Web Services. We have to deal with some specific aspects of these systems, like authentications, intrusion, sniffing, etc. as well as with the classical problems related to concurrency, such as non-determinism, parallelism, etc. All of these aspects will be covered throughout this dissertation.

Security aspects concerning e-Commerce and e-Business applications must be also taken into account. Enormous efforts have been spent to solve these problems, but no final solution has been fixed yet, and the research community shares the viewpoint that it is not possible to achieve a master key.

Furthermore in many cases, not only the classical aspects of concurrency mentioned above must be considered, but also some quantitative aspects, like time or probabilities. For example, there are e-Commerce sites that need a response before a date, if the response has not arrived on time, then an error occurs and the costumer will not probably receive his purchase.

To sum up, the objective in this dissertation is the application of formal

methods to Internet based applications, i.e., the application of formal description languages and Model Checking in the development of Web Services.

1.3 Contributions

The main contributions of this dissertation revert in the area of the application of formal methods to the emerging Internet techniques. The areas where Model Checking technique has been applied include security protocols such as "Transport Layer Security" (TLS), "Security Socket Layer" (SSL) and also to "Secure Electronic Transaction" (SET), as well as the emerging Internet applications, "Web Services". Concerning Internet Security Protocols the main aspects to deal with have been the security and time aspects. And the effort spent in Web Services have focused on the development of a verification methodology, which applies to the highest layers of the Web Services architecture (Choreographes and Orchestrations where the abstract level is suitable for using Formal Methods).

1.4 Dissertation Structure

This dissertation is structured in six chapters. This chapter has presented an overview, and in chapter 2 we will show the state of the art of this line of research. Here we review the formal methods and techniques used in the development and verification of Internet applications, and more specifically, in the development of Security Protocols and Web Services.

In the third chapter we show the Model Checking Technique in deep and how it can be applied to Real Time Systems and Probabilistic Systems. In this chapter we also review three tools: UPPAAL, P_UPPAAL and RAPTURE, which are very useful for simulating and analysing this kind of system.

Chapter 4 contains the application of Model Checking Techniques to e-commerce security protocols, and chapter 5 presents a methodology for the verification of Web Services, by using Model Checking Techniques.

Finally, chapter 6 is devoted to the conclusions and future works.

CHAPTER 2

Formal Methods and Internet Based Systems

Formal methods, and the development of methodologies based on these formalisms can yield significant benefits in the development of e-business systems, due to the enhanced rigor these methods bring.

In this chapter we describe the formal techniques that we use throughout the dissertation, and related work of the specific area of application considered. Concretely, we apply formal methods to Internet applications, and more specifically, to e-commerce and Web Services.

But first we must answer the following questions: 1) Why are Formal Methods needed? 2) What kind of systems need to be verified? and 3) How are Formal Methods to be applied to these systems?

The need for Formal Methods has been shown in many industrial cases, where their absence has sometimes led to catastrophic results, either in economic terms or even in human loss, as in the crash of the Ariane 5 [102]. With respect to the second question, formal methods can be applied to a wide range of systems, although we are mainly interested here in distributed systems, for which some specific techniques can be applied. In order to apply these techniques it is important to fix the adequate level of abstraction, to describe only

those aspects of the system that are relevant.

The application of these techniques heavily depends on the type of system being studied, and the particular methodology used. In the case of real-time systems, for instance, these techniques must be applied with severe correctness requirements, and the property class of interest are focused on time or probabilistic issues.

2.1 Formal Methods

The application of formal techniques [138, 15] to the development of computer systems provides mathematically based techniques that describe the system behavior. These methods therefore provide a framework for systematically specifying, developing and verifying systems.

A method regarded as formal has a sound mathematical basis and this mathematic rigor can be introduced by using several concepts, but typically it is given by a specification language. The specification language can define the real system providing a formal model of the system and the model can precisely define notions such as consistency, completeness, implementation and correctness. Furthermore, it can also define the properties that the system must satisfy in order to achieve the above notions.

This mathematical rigor bring us some advantages over informal systems, for instance: 1) Knowing the specification is realizable, 2) Knowing the system has been developed correctly, 3) The possibility of checking properties (without necessarily running the system in order to determine its behavior) and, 4) Generating tests, etc.

These advantages can be applied at any stage of computer system development and at each stage it provides different information. For example, in the customer's requirement stage, we can specify the properties that the system must fulfil. These properties could be used in the next stages to check the specification model (at the design stage), and even the real system (at the test and debugging stages).

One tangible product of applying formal methods to a computer system is a formal specification (a formal model). A specification serves as a contract,

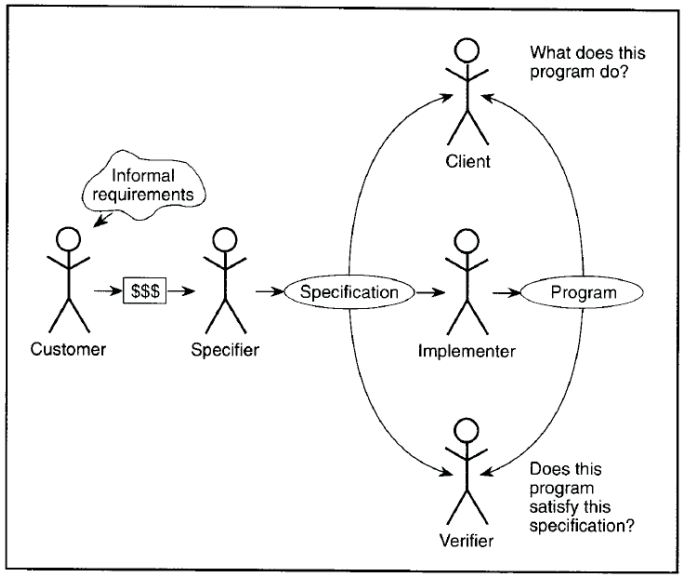


Figure 2.1: Specification users.

a piece of documentation and a means of communication among a client, a specifier and an implementer, as shown in Figure 2.1.

Another characteristic of formal methods is that they are complex methods whose application could become unfeasible without some kind of tool support. In this sense, two key constraints for tool implementation must be taken into account: “The language syntax of the specification must be explicit" and “the language semantic of the specification must be restricted". Thus, formal specifications have the additional advantage over informal ones of being amenable to machine analysis and manipulation.

In the next subsections we will discuss the need for formal methods as well as their basis, application, classification and other issues, including the model checking technique, which is the central formal technique used in this dissertation.

2.1.1 The Need for Formal Methods

The kind of system where software and hardware have been used covers a wide range of applications, where no failure is acceptable: e-commerce, mobile network technology, highway, subway, train and air control systems, medical instruments, weapon technology, nuclear energy, and a long list of etceteras. There have been several media reports on incidents that have been caused by an error in software and hardware systems. Recent examples of this kind of failures are the failures in aerospace transport, causing loss of life and damage amounting to hundreds of billions of euros.

It is clear, then, a totally secure system where the hardware and software command critical areas must be ensured, because these systems are becoming a more and more integral part of our lives on a daily basis. A possible solution would be to dismantle these systems to avoid the failures, although it has been demonstrated that manual control is less risk-free than the automatic software and hardware and it would cause a socio-economical upheaval.

Because of the success of the Internet and 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 most likely accelerate in coming years. Because of this rapid growth in technology, it will become even more important to develop methods that increase our confidence in the correctness of such systems.

2.1.2 Formal Specification Languages

Let us to introduce readers in the basis of formal methods, then, our next task is to explain the basis of formal methods to our readers. We first begin with the definition of a formal method, by defining the language in which the formal method is written.

A formal specification language provides a formal method's mathematical basis. It provides a notation (its syntactic domain), a universe of objects (its semantic domain) and a precise rule defining the objects that satisfy each specification. A specification is a sentence written in terms of elements of the syntactic domain and denotes a specific subset of the semantic domain.

If we borrow the following terms and definitions from Guttan el al [63], we can rigorously define the above definition:

Definition 1. *A formal specification language is a triple,* $< Syn, Sem, Sat >$*, where* Syn *and* Sem *are sets and* $Sat \subseteq Syn \times Sem$ *is a relation between them.* Syn *is called the language's syntactic domain;* Sem*, its semantic domain; and* Sat*, its satisfied relation.*

Definition 2. *Given a specification language,* $< Syn, Sem, Sat >$*, if* $Sat(syn, sem)$ *then* syn *is a specification of* sem*, and* sem *is a specificand of* syn*.*

Definition 3. *Given a specification language* $< Syn, Sem, Sat >$*, the specificand set of a specification* syn *in* Syn *is the set of all specificands* sem *in* Sem *such that* $Sat(syn, sem)$*.*

By using these definitions, we can define, for example, when a program satisfies a given specification:

Definition 4. *Given a specification language* $< Syn, Sem, Sat >$*, an implementation* $prog$ *in* Sem *is correct with respect to a given specification* $spec$ *in* Syn *if and only if* $Sat(spec, prog)$*.*

Furthermore, we can check if our specifications satisfy certain properties as, for example, whether each well-formed specification is unambiguous:

Definition 5. *Given a specification language* $< Syn, Sem, Sat >$*, a specification* syn *in* Syn *is unambiguous if and only if* Sat *maps* syn *to exactly one specification set.*

Another desirable property of a specification should be, consistency:

Definition 6. *Given a specification language* $< Syn, Sem, Sat >$*, the a specification* syn *in* Syn *is consistency (or satisfiable) if and only if* sat *maps* syn *to a non-empty specificand set.*

Most formal methods are defined in terms of a specification language that has a well-defined logical inference system. A logical inference system defines a consequence relation, typically given in terms of a set of inference rules, mapping a set of well formed sentences in the specification language to a set of

well-formed sentences. Thus, we can use this inference system to prove properties from the specification about specificands.

As we have a clear idea of how formal methods work, the next step is to show where they can be applied.

2.1.3 Applying Formal Methods

As we have mentioned above, the use of formal methods is extended to all phases of computer system development. However, we should not consider these applications as separate, but rather as an integral activity.

Requirement Analysis

The main advantage of applying formal methods in this phase is that formal methods help clarify the customer's set of informally stated requirements. A formal specification helps concretize and standardize the customer's vague ideas. Furthermore, it also helps reveal contradictions, ambiguities and incompleteness of the customer's requirements. We can find some examples in the bibliography such as kate [4], the Requirement Apprentice [121], the Gist [130], KAOS [89], etc.

System Design

Key elements in this design phase are the decomposition and refinement. The decomposition is the process of partitioning a problem in subproblems or modules where the degree of difficulty has obviously decreased. The refinement also performs a decomposition, but it refers to the level of abstraction used in the system specification. Works such as Dijkstra [54] and Hoare [74, 76, 75], languages such as Refine and Extend ML [126] and support tools such as CIP-S [9], Ergo Support System [98] and Nuprl [36] are used in this phase.

System Verification

The verification process checks whether the system satisfies its specifications or not. The formal verification is available only when the system has been formally

specified. Due to the real system being unbounded, the verification process cannot accomplish an entirely system verification, although it can accomplish the verification of critical areas. Thus a key element in this phase is also the abstraction level where the verification process is applied. System such us Gypsy [18], The Hierarchical Development Method (HDM) [100], the Formal Development Method (FDM) [37], m-EVES (Environment for Verifying and Evaluating Software) [40], Higher Order Logic (HOL) [60], etc. perform system verification.

System Validation

Formal methods aids the testing and debugging of systems by using them as test generators for black-box testing. Furthermore, they can be used in other kind of testing such as path testing, unit testing and integration testing. Systems such as Data Abstraction, Implementation, Specification, Testing Systems [108], Kremmerer's symbolic execution tool [86] and Task Sequencing Language Runtime System [124] perform the system validation.

System Documentation

A key document for Documentation process is a precise and standardized description of the computer system. This definition accomplishes the description of a formal specification. Furthermore, this document also helps in the communication process among customers specifiers, implementers and verifiers, as we have shown in figure 2.1.

System Analysis and Evaluation

Once the system has been implemented and tested, it is mandatory to perform an analysis and an evaluation of the implementation functionality and performance. An important question that this analysis must resolve is whether the implementation satisfies the initial customer's expectations or not.

2.1.4 A Partial Taxonomy

Formal methods can be classified by a characteristic list that would be too long to cite in its entirely. For example, we could classify them according to whether

the type of underlying logic is first order or not, or by using the kind of language used (visual or not), etc. The purpose is not to perform an exhaustive taxonomy, but to proffer a classification of the most widely known formal methods and techniques.

Model-oriented and Property-oriented

A widely extended taxonomy of formal methods is the classification according to orientation. This classification divides formal methods into those that capture the system behaviors in a formal model and those that derive the system behaviors from the formal properties that the system must satisfy. However, in some cases there are formal techniques that have both characteristics.

Model-oriented methods capture the system behaviors in a model written by using a formal specification. This subclassification includes Parna's state machines [113], Robinson and Robine's extensions [122], VDM, Z [69], Petri Nets [114], Milner's Calculus of Computational Systems [109], Hoare's Communicating Sequential Processes (CSP) [76], Unity [29] , timed automata [2], CCS [71], etc.

Property-oriented methods can be subclassified into two categories, axiomatic and algebraic. Axiomatic methods are based on Hoare's work on correctness proofs of implementations of abstract data types [74, 75]. Some examples of axiomatic methods are Iota [110], OBJ [59], Anna [106] and Larch [62]. Algebraic methods define as heterogeneous algebras both elements, data types and processes. A great deal of work has been done on algebraic specification, although the most representative are probably Clear [25] and Act one [57] (Algebraic Specification Techniques for Correct and Trusty Software Systems), extensions to the Hoare-axiom method, temporal logic [115], Lamport's transition axiom method [88], and LOTOS [14].

Visual Languages

This classification includes any formal method whose specification language can be represented in a graphic manner. One of the most representative for this type of formal method are the Petri Nets and their variations. However, nowadays, since the greater part of support tool interfaces have become integrated into

graphic environments, their formal specification can be depicted in a visual manner. The inclusion of visual information in some cases has actually turned those methods into informal ones, because of the integration of informal data into the visual specifications.

Some examples of these methods are Petri Nets, State Transition Diagrams, Harel's state chart [66], Miro visual languages [73], and HIPO [128].

Executable

There are some examples of formal methods whose specification supports the execution on a computer. This kind of formal method is more restrictive, because the specification must be computable and its domain finite or have a finite representation. Some examples are OBJ, PROLOG [35], and Paisley [141].

Tool-supported

Nowadays, formal methods in general provide the support of a tool to manipulate the specifications and programs. A clear example of this kind of technique is Model Checking, which we will see later in more detail. Model Checking tools let users construct a finite-state model of the system and then show how a property holds in the states or transition. Another example of tool-supported technique is proof-checking. These tools let the user treat algebraic specifications as rewrite rules. It includes Affirm [132], REVE [99], the Larch Prover, Boyer-Moore Theorem Prover [30], FDM, HDM, m-EVES [61], HOL, LCF [90], OBJ, and others.

2.1.5 Some Examples

Temporal Logic

Temporal logic is a property-oriented method for specifying properties of concurrent and distribute systems. For a given temporal logic inference system, special modal operators concisely state assertions about system behavior. Specifiers use these operators to refer to past, current, and future states or events.

There is no one standard temporal logic inference system nor one standard set of operators. Modal operators commonly used are $\Diamond$, $\Box$ and $\bigcirc$. Informally,

$$\langle right!m\rangle \Rightarrow \underline{\diamondsuit}\ \langle left!m\rangle \quad (1)$$

$$(\langle right!m\rangle \wedge \ominus \underline{\diamondsuit}\ \langle right!m'\rangle) \Rightarrow \underline{\diamondsuit}\ (\langle left!m\rangle \wedge \ominus \underline{\diamondsuit}\ \langle left!m'\rangle) \quad (2)$$

$$\langle right!m\rangle \Rightarrow \underline{\diamondsuit}\ \langle left!m\rangle \quad (3)$$

$$\langle right!m\rangle \Rightarrow \underline{\diamondsuit}\ \langle left!m\rangle \quad (4)$$

Figure 2.2: Temporal logic specification of an unbounded buffer.

when interpreted with respect to a sequence of states $\Box P$ says that in all future states, the stated predicate P holds; $\diamond P$ says that in some future state, P will hold; and $\bigcirc P$ says that in the next state P will hold. For example, $P \Rightarrow \diamond Q$ says that if P holds in the current state, Q will eventually hold. Temporal logic notation tends to be terse and temporal logic specifications are simply an unstructured set of predicates, all of which must be satisfied by a given implementation.

Figure 2.2 presents a temporal logic specification of the behavior of an unbounded buffer in an asynchronous environment. The example is adapted from Koymans el al [87], using the temporal logic system in Pnueli [116], which has 12 modal operators. The formulas are interpreted with respect to sequences of events. A buffer has a left input channel and a right output channel. The expression $c!m$ denotes the event of placing message m on channel c. The first predicate states that any message transmitted to the right channel ($\langle right!m\rangle$) must have been previously placed on the left channel $\underline{\diamondsuit}\ \langle left!m\rangle$. The second predicate states that messages are transmitted first in, first out. The third predicate, states that all messages are unique. And the last predicate states that all incoming message will eventually be transmitted. The three first predicates are safety message whereas the last one is a liveness property.

CSP

CSP uses a model-oriented method for specifying a concurrent process and a property-oriented method for stating and proving properties about the model.

$BUFFER = P_{<>}$
where $P_{<>} = left?m \rightarrow P_{<m>}$
and $P_{<m>^\frown s} = (left?n \rightarrow P_{<m>^\frown s^\frown <n>} | rigth!m \rightarrow Ps)$

BUFFER **sat** $(right \leq left) \wedge (\textbf{if } right = left \textbf{ then } left \notin ref \textbf{ else } right \in ref)$

Figure 2.3: CSP program and specification of an unbounded buffer.

CSP is based on model of traces, or event sequences and assumes that processes communicate by sending messages across channels. Processes synchronize on events so the event of sending output messages m on named channel c is synchronized with the event of simultaneously receiving an input message on c. Figure 2.3 gives a CSP specification of an unbounded buffer. BUFFER itself is specified to be a process P that acts as an unbounded buffer. The recursive definition of P is divided into two clauses to handle the empty and non-empty cases. The first clause, says that if the buffer is empty, in the event that there is a message m on the left channel ($left?m$), it will input it. In CSP, if x is an event and P is a process, the notation $x \Leftarrow P$ denotes a process that first engages in the event x and then behaves exactly as described by P. The second clause says that if the buffer is non-empty, then either the buffer will input another message n from the left channel, appending it to the end of the buffer, or output the first message in the buffer to the right channel.

In CSP's formalism, BUFFER is a CSP program; we can state and prove properties about the traces it denotes. Using algebraic laws on traces, we can formally verify that a given CSP program satisfies a specification on traces. The last line in figure 2.3 states that a BUFFER describes a set of traces, each of which satisfies the predicate given on the right side of sat.

2.1.6 Model Checking

Model checking is a verification technique that explores all possible system states in a brute force manner [32, 33]. Similar to a computer chess program that checks all possible moves, a model checker, the software tool that performs the model checking, examines all possible system scenarios in a systematic manner.

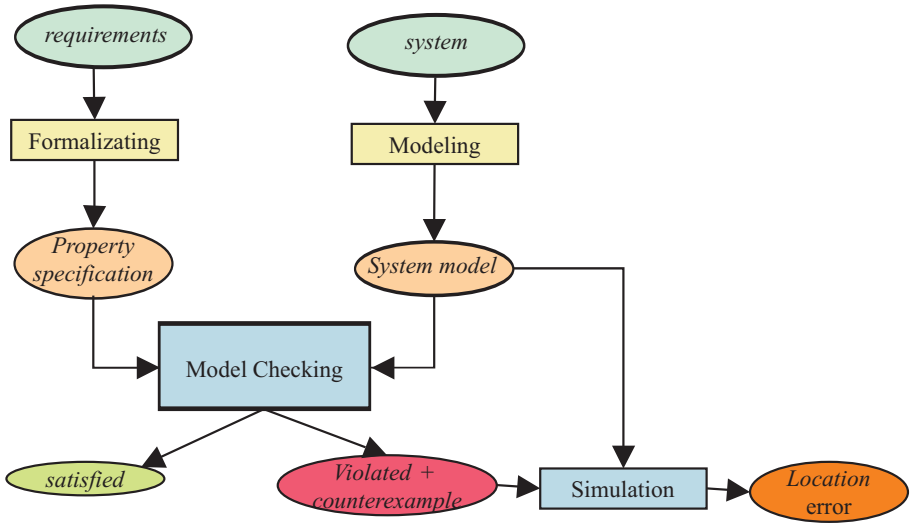


Figure 2.4: Schematic view of the model-checking approach

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 and 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 [107] or Java [56] or hardware description languages such as Verilog [131] or VHDL [78]. The model checker examines all relevant systems states to check whether they satisfy or not the desired property. If a state is encountered that violates the property under consideration, the model checker provides a counterexample with an execution path leading 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, also obtaining some useful debugging information, which permits to adapt the model or the property accordingly (see Figure 2.4).

Characteristics of Model Checking

In [84] we can find a good definition of what model checking is:

> 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.

The development of Model Checking

Applying model checking to a design consists of several tasks, each of which is 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 time and memory, the modeling of a design may require the use of abstractions to remove irrelevant or unimportant details.

Specification Before verification, it is necessary to state the properties that the design must satisfy. For this purpose, 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 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 the event 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 from 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.

Temporal Logic and Model Checking

Before considering the advantages and drawbacks of model checking, we explain in more detail temporal logics and how they are used in model checking [32, 33].

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 future. 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^*. This is a very expressive logic that combines both branching-time and linear-time operators [31].

Advantages and Drawbacks of Model Checking

The main advantages of model checking are the following:

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 us to focus first on the essential properties. Thus, no complete requirement specification is needed.

3. It is not vulnerable to the likelihood 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 the event of a property being invalidated; this is very useful for debugging purposes.

5. It is a potential “push-button" technology; the use of model checking requires neither 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* into existing development cycles; its learning curve is not very steep, and 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 suited to *control-intensive* applications as data typically range over infinite domains.

2. Its 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,* not the system itself. In consequence any result obtained is for 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. Its usage requires some *expertise* in finding appropriate abstractions to obtain smaller system models and to state properties in the logical formalism used.

7. There is no guarantee to obtain correct results: a model checker may also contain *software defects.*

8. It does not allow *generalizations* to be checked: in general, checking systems with an arbitrary number of components, or parameterized systems cannot 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 for exposing potential design errors.

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

In the next chapter we will describe the UPPAAL tool which includes a model checker.

2.2 Internet Based Systems: e-Commerce, Security Protocols and Web Services

In this section we describe the Internet-based contexts to which the formal techniques presented in this dissertation are applied. These different contexts are not separate areas, but rather areas that interrelate with each other. For example, e-commerce and Web Services systems can use security protocols in their communications, or e-commerce systems can be developed by using Web Services.

2.2.1 e-commerce

Electronic commerce (e-commerce) became a buzzword as the information society developed rapidly throughout the 1990s. Internet has made e-commerce available to a wider user group, notably smaller enterprises and households. Amongst the business community the search for increased productivity and efficiency is expected to lead to even more enterprises adopting e-commerce as a way of doing business in the future. Whilst an ever growing awareness of the opportunities, technological developments in infrastructure and access devices, as well as falling access costs will facilitate this, fears about security and a lack of skills could hold it back.

Thus, the economic importance of e-commerce has grown every year. In Spain, for instance, the economic impact of e-commerce in the last decade is remarkable (15% of the total Gross Domestic Product Increment from 1995 to 2002). In contrast, 70% of the population thinks that Internet, and consequently e-commerce, are not necessary, and half of the population of Spain cannot access Internet.

A widespread opinion that has helped to create this situation is that software is not free of errors. This means that errors in Internet can cost a lot of money, efforts and resources. For example, in e-Commerce a security hole in the communication process could let a cracker obtain the credit card information of a customer. Thus, the Internet community response to these attacks has been the development of protocols that avoid the intrusion. We discuss some examples of these protocols in the next subsection.

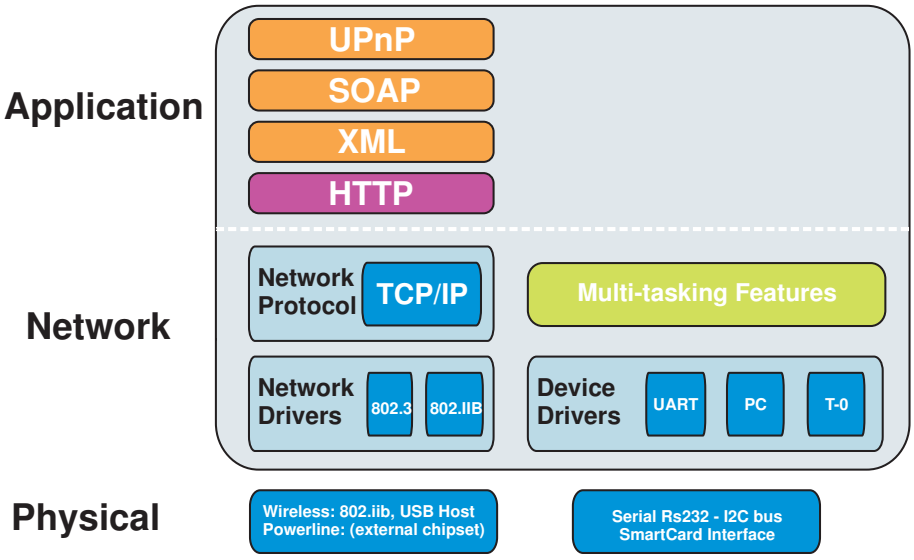


Figure 2.5: The Internet Layers and Protocols

Figure 2.5 shows the internet layers and protocols. In this figure the protocols are divided into three layers: Physical layer, Network layer and Application layer. The Physical layer involves the protocols used for transmitting information between the physical devices that support this technology (wireless, ethernet, optical, etc). The Network layer consists of the protocols that deal with the problem of partitioning and switching the information that must be transmitted, a clear example being the TCP/IP [129] protocol. The Application layer involves any protocol or application using the network layer as communication media. e-Commerce applications run over the application layer, so that they can be developed by using the protocols of this layer as for example HTTP, HTTPS, XML, etc. Web Services are a growing technology of this Application layer. This technology uses application protocols such as SOAP, XML and HTTP.

2.2.2 Security Protocols

Security is, without question, one of the most important issues to be considered on Internet applications. The Internet research community has provided Internet users with some security protocols to avoid intruder access. These protocols have enforced the confidence of users in Internet applications, and as a result they are starting to use them to make purchases, access bank account information, organize theirs trips, contact with government offices and in general deal with daily activities in order to improve their life quality.

Security Protocols are usually divided into two types. The first type allows the Internet Session to be secure in terms of intrusion, while the second type ensures the security of some confidential information like the credit card number.

Security Protocols are based on a paired protocol separated into two phases. The first phase allows the authentication of the parties involved in the transaction and the negotiation of cryptographic information. The second phase uses the cryptographic parameters fixed during the first phase to encrypt the information.

There are a lot of security protocols that implement the security concerns in Internet. We now describe three particular protocols:

TLS and SSL

The Transmission Control Protocol/Internet Protocol (TCP/IP) governs the transport and routing of data over Internet. Other protocols, such as the HyperText Transport Protocol (HTTP), Lightweight Directory Access Protocol (LDAP), or Internet Messaging Access Protocol (IMAP), run "on top of" TCP/IP, in the sense that they all use TCP/IP to support typical application tasks such as displaying web pages or running email servers.

According to figure 2.6, the Transport Layer Security protocol [58], TLS for short, and the Secure Socket Layer, SSL for short, run above TCP/IP and are integrated into the network layer. This layer appears below the application layer , in which we can find protocols like HTTP or IMAP. It uses TCP/IP on behalf of the higher-level protocols, and allows a TLS- or SSL-enabled server to authenticate itself to TLS or SSL enabled clients. It also allows the client to authenticate himself to the server, and it allows both machines to establish an

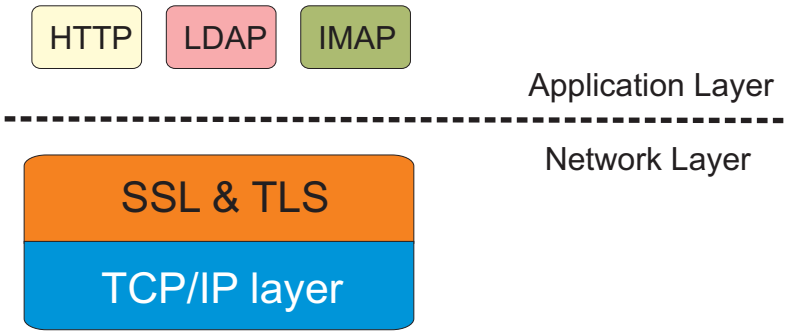


Figure 2.6: TLS runs above TCP/IP and below high-level application protocols

encrypted connection. These capabilities address fundamental concerns about communication over Internet and other TCP/IP networks.

TLS and SSL server's authentication certify allows a user to confirm a server's identity. TLS or SSL enabled client software can use standard techniques of public-key cryptography to check that a server's certificate and public ID are valid and have been issued by a certificate authority (CA) listed in the client's list of trusted CAs. This confirmation might be important if the user, for example, is sending a credit card number over the network and wants to check the receiving server's identity.

SET

Secure Electronic Transaction (SET) [134] is a system for ensuring the security of financial transactions on the Internet. It was supported initially by Mastercard, Visa, Microsoft, Netscape, and others. With SET, a user is given an electronic wallet (digital certificate) and a transaction is conducted and verified using a combination of digital certificates and digital signatures between the purchaser, a merchant, and the purchaser's bank in a way that ensures privacy and confidentiality. SET makes use of Netscape's Secure Sockets Layer (SSL), Microsoft's Secure Transaction Technology (STT), and Terisa System's Secure Hypertext Transfer Protocol (S-HTTP). SET uses some but not all of the aspects of a public key infrastructure (PKI), and works in the following way:

Assuming that a customer has a SET-enabled browser, such as Netscape or

Microsoft's Internet Explorer, and that the transaction provider (bank, store, etc.) has a SET-enabled server, a transaction is made according to the following steps:

1. The customer opens a Mastercard or Visa bank account. Any issuer of a credit card is some kind of bank.

2. The customer receives a digital certificate. This electronic file functions as a credit card for online purchases or other transactions. It includes a public key with an expiration date, which has been validated by the bank by using a digital switch.

3. Third-party merchants also receive certificates from the bank. These certificates include the merchant's public key and the bank's public key.

4. The customer places an order over a Web page.

5. The browser of the customer receives and confirms that the merchant is valid from its certificate.

6. The browser sends the order information. This message is encrypted with the merchant's public key, the payment information, which is encrypted with the bank's public key (which cannot be read by the merchant), and information that ensures the payment can only be used with this particular order.

7. The merchant verifies the customer by checking the digital signature on its certificate. This may be done by referring the certificate to the bank or to a third-party verifier.

8. The merchant sends the order message to the bank. This includes the bank's public key, the customer's payment information (which the merchant cannot decode), and the merchant's certificate.

9. The bank verifies the merchant and the message. The bank uses the digital signature on the certificate with the message and verifies the payment part of the message.

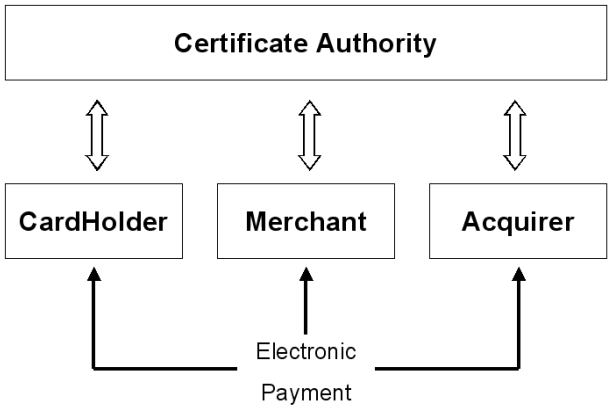


Figure 2.7: Participants in a SET protocol and their Interactions.

10. The bank digitally signs and sends authorization to the merchant, who can then fill the order.

Figure 2.7 represents the main participants in a SET protocol (the cardholder, the merchant and the acquirer) and the relations between them. Thus, double arrows represent the relation with the certificate authority who issues the digital certification in order to authenticate each involved party, while single arrows stand for the process of electronic payment.

This figure depicts the main participants, although there are other participants such as the issuer, the payment gateway and the brand. All the institutions and parties involved in this protocol are now defined:

Cardholder A cardholder uses a payment card that has been issued by the Issuer. SET ensures that the interactions of cardholders with the merchants and the payment card information remain confidential.

Issuer This is a financial institution. It establishes an account for the cardholder. It also issues the payment card. The Issuer also guarantees payment for authorized transactions using the payment card.

Merchant The merchant offers goods or services in exchange for a payment. In order to accept payment cards the merchant must have a relationship with an Acquirer.

Acquirer A financial institution that establishes an account with a merchant and process payment card authorizations and payments.

Payment Gateway It is a device operated by an Acquirer or a designated third party that processes merchant payment messages, including payment instructions from cardholders.

Brand Financial Institutions come up with payment card brands in order to protect and advertise the brand. It creates an atmosphere conducive to establishing and enforcing rules for use and acceptance of their respective payment cards. It also provides a network to interconnect the financial institutions.

Third Parties Issuers and Acquirers can assign processing of payment card transactions to these third-party processors.

2.2.3 Web Services

Nowadays, a key layer for Internet is the Application layer and in recent years some new techniques and languages for developing distributed applications over the Internet have appeared, such as the Extensible Markup Language, XML, and some new Web Services frameworks [34, 64, 137] for describing interoperable data and platform neutral business interfaces, enabling more open business transactions to be developed.

Web Services are a key component of the emerging, loosely coupled, Web-based computing architecture. A Web Service is an autonomous, standards-based component whose public interfaces are defined and described using XML [85]. Other systems may interact with a Web Service in a manner prescribed by its definition, using XML-based messages conveyed by Internet protocols.

The Web Services specifications offer a communication bridge between the heterogeneous computational environments used to develop and host applications. The future of E-Business/E-Commerce applications requires the ability to perform long-lived, peer-to-peer collaborations between the participating services, within or across the trusted domains of an organization.

The Web Service architecture stack targeted for integrating interacting applications consists of the following components:

- **SOAP[64]:** This defines the basic formatting of a message and the basic delivery options regardless of programming language, operating system, or platform.
- **WSDL[137]:** This describes the static interface of a Web Service. At this point the message set and the message characteristics of end points are here defined. Data types are defined by XML Schema specifications.
- **Registry[34]:** This makes an available Web Service visible, and describes the concrete capabilities of a Web Service.
- **Security layer:** This ensures that exchanges of information are not modified or forged in a verifiable manner and that parties can be authenticated.
- **Reliable Messaging layer:** This provides a reliable layer for the exchange of information between parties.
- **Context, Coordination and Transaction layer:** This defines interoperable mechanisms for propagating the context of long-lived business transactions and enables parties to meet correctness requirements by following a global agreement protocol.
- **Business Process Languages layer[5, 6]:** This describes the execution logic of Web Services based applications by defining their control flows (such as conditional, sequential, parallel and exceptional execution) and prescribing the rules for consistently managing their non-observable data.
- **Choreography layer[85]:** This describes collaborations of parties by defining from a global viewpoint their common and complementary observable behavior where information exchanges occur, when the jointly agreed ordering rules are satisfied.

Choreography Layer - WSCDL Description

The Web Services Choreography specification is intended to offer precise descriptions of collaborations between any type of party, regardless of the supporting platform or programming model used by the implementation of the

hosting environment. Using the Web Services Choreography specification, we produce a contract containing a “global" definition of the common ordering conditions and constraints under which messages are exchanged. This contract describes, from a global viewpoint, the common and complementary observable behavior of all the involved parties. Each party can then use the global definition to build and test solutions that conform to it. The global specification is in turn realized by a combination of the resulting local systems, on the basis of appropriate infrastructure support.

In real-world scenarios, corporate entities are often unwilling to delegate control of their business processes to their integration partners. Choreographies offer a means by which the rules of participation within a collaboration can be clearly defined and agreed to, jointly. Each entity may then implement its portion of the Choreography as determined by the common or global view. It is the purpose of WS-CDL that the conformance of each implementation to the common view expressed in WS-CDL is easy to determine. Figure 2.8 demonstrates a possible usage of the Choreography Description Language, where we see that we use WS-BPEL as the Business Process Execution Layer (BPEL for short).

WS-CDL describes interoperable collaborations between parties. In order to facilitate these collaborations, services commit themselves to mutual responsibilities by establishing Relationships. Their collaboration takes place in a jointly agreed set of ordering and constraint rules, whereby information is exchanged between the parties. The WS-CDL model consists of the following entities:

- **Participant Types, Role Types and Relationship Types** within a Choreography. Information is always exchanged between parties within or across trust boundaries. A Role Type enumerates the observable behavior a party exhibits in order to collaborate with other parties. A Relationship Type identifies the mutual commitments that must be made between two parties for them to collaborate successfully. A Participant Type groups together those parts of the observable behavior that must be implemented by the same logical entity or organization.

- **Information Types, Variables and Tokens.** Variables contain information about commonly observable objects in a collaboration, such as

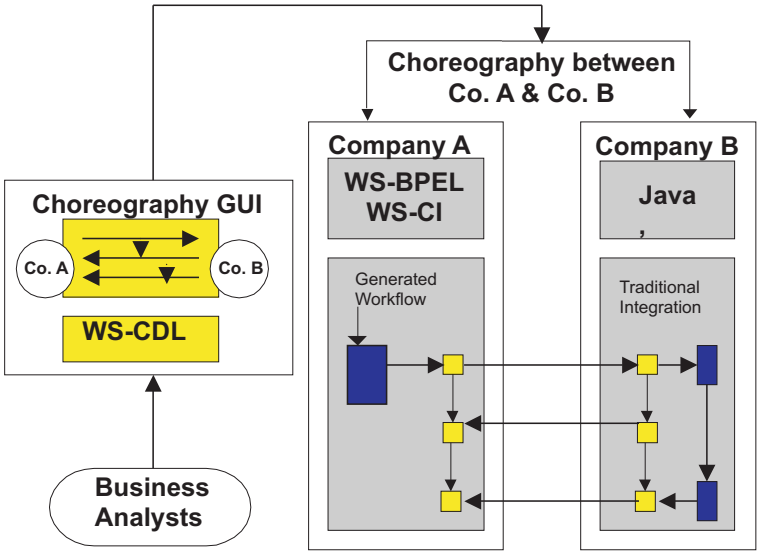


Figure 2.8: WS-CDL and WS-BPEL usage.

the information exchanged or the observable information of the Roles involved. Tokens are aliases that can be used to reference parts of a Variable. Both Variables and Tokens have Types that define the structure of what the Variable contains or the Token references.

- **Choreographies** define collaborations between interacting parties:
  - **Choreography Life-line**: This shows the progression of a collaboration. Initially, the collaboration is established between the parties; then, some work is performed within it, and finally it completes either normally or abnormally.
  - **Choreography Exception Block**: This specifies the additional interactions that should occur when a Choreography behaves in an abnormal way.
  - **Choreography Finalizer Block**: This describes how to specify additional interactions that should occur to modify the effect of an earlier successfully completed Choreography (for example to confirm or undo the effect).

- **Channels** establish a point of collaboration between parties by specifying where and how information is exchanged.
- **Work Units** prescribe the constraints that must be fulfilled for making progress and thus performing actual work within a Choreography.
- **Activities and Ordering Structures.** Activities are the lowest level components of the Choreography that perform the actual work. Ordering Structures combine activities with other Ordering Structures in a nested structure to express the ordering conditions in which information within the Choreography is exchanged.
- **Interaction Activity** is the basic building block of a Choreography, which results in an exchange of information between parties and possible synchronizations of their observable information exchanges, and the actual values of the exchanged information.

Figure 2.9 shows a part of a choreography specification. The example used in this specification describes the collaborative behavior of a buyer, a seller, a credit check agency and a shipper. This figure specifies the interaction when the buyer requests a quote, then a message is sent from the buyer to the seller with the details of the product quoted. The seller replies with a quote back to the buyer.

Orchestration Layer - WS-CI and WS-BPEL Descriptions

WSCI is an interface description language, that describes the observable behavior of a service and the rules for interacting with the service from outside. In a WSCI specification, the observable behavior of each party in a message exchange is described independently of the other involved parties. It is not an executable language, although it is precise and unambiguous enough.

The basic construct of WSCI is the Action, which is bound to some WS-CDL operation.

The main concepts in WSCI language are the following:

Interface .- WSCI maps the description of a Web Service to the notion of an interface.

```
 <interaction name="QuoteElicitation"
operation="getQuote" channelVariable="tns:Buyer2SellerC">
    <description type="documentation">
        Quote Elicitation
    </description>
    <participate relationshipType="tns:Buyer2Seller"
                 fromRoleTypeRef="tns:BuyerRole"
                 toRoleTypeRef="tns:SellerRole"/>
    <exchange name="QuoteRequest"
              informationType="tns:QuoteRequestType"
              action="request">
        <description type="documentation">
                Quote Request Message Exchange
        </description>
        <send variable=
                "cdl:getVariable('quoteRequest','','')"/>
        <receive variable=
                "cdl:getVariable('quoteRequest','','')"/>
    </exchange>
    <exchange name="QuoteResponse"
              informationType="tns:QuoteResponseType"
              action="respond">
        <description type="documentation">
            Quote Response Message Exchange
        </description>
        <send variable=
                "cdl:getVariable('quoteResponse','','')"/>
        <receive variable=
                "cdl:getVariable('quoteResponse','','')"/>
    </exchange>
</interaction>
```

Figure 2.9: Part of a choreography specification.

Activities and choreography of activities .- WSCI describes the behaviour of a Web Service in terms of choreographed activities. A choreography describes temporal and logical dependencies among activities, where atomic activities represent the basic unit of behavior of a Web Service.

Processes and units of reuse .- A process is a portion of behavior labeled with a name. It can be reused by referencing its name.

Properties .- These allow us to reference a value within the interface definition. They are the equivalent of variables on other languages.

Context .- This describes the environment in which a set of activities is executed. Each activity is defined in exactly one context definition.

Message correlation .- This describes how conversations are structured and which properties must be exchanged to perform the service correctly.

Exceptions .- The definition of exception is part of the context definition. There are different types of exceptions and when a exception occurs the current context must terminate after the activities associated with the exception have been performed.

Transactions and compensation activities .- A transaction asserts that a set of activities is executed in an all-or-nothing way. A transaction may declare a set of compensation activities that will be executed if the transaction has completed successfully, but needs to be undone.

Global model .- The global model is described by a collection of interfaces of the participating services and a collection of links between the operations of communicating services.

In Figure 2.10 we can see a WS-CI example that describes part of a travel agent specification. This specification is part of a travel reservation system example that describes a reservation system for airlines, travel agents and travelers. This detail defines an exception to handle the traveler's reservation timeout.

WS-BPEL is also an interface description language, which describes the observable behavior of a service by defining business processes consisting of

```
...
<context>
 <process name ="BookSeats" instantiation="other">
  <action name="bookSeats"
          role="tns:travelAgent"
          operation="tns:TAtoAirline/bookSeats">
  </action>
 </process>
 <exception>
  <onMessage>
   <action name="ReservationTimeOut"
         role="tns:TravelAgent"
         operation="tns:TAtoAirline/AcceptCancellation">
       <correlate
           correlation="defs:reservationCorrelation"/>
   </action>
   <action name="NotifyOfTimeOut"
     role="tns:TravelAgent"
     operation="tns:TAtoTraveller/NotifyofCancellation"/>
     <fault code="tns:reservationTimedOut"/>
  </onMessage>
  ...
 </exception>
  ...
</context>
```

Figure 2.10: Part of a Travel Agent WS-CI Specification

stateful long-running interactions in which each interaction has a beginning, a defined behavior and an end, all of this being modeled by a flow, which consists of a sequence of activities. The behavior context of each activity is defined by a scope, which provides fault handlers, event handlers, compensation handlers, a set of data variables and correlation sets.

Let us now see a brief description of these components:

- **Events**, which describe the flow execution in an event-driven manner.
- **Variables**, which are defined by using WSDL schemes, for internal or external purposes, and are used in the message flow.
- **Correlations**, which identify processes interacting by means of messages.
- **Fault handling**, defining the behavior when an exception has been thrown.
- **Event handling**, defining the behavior when an event occurs.
- **Activities**, which represent the basic unit of behavior of a Web Service. In essence, WS-BPEL describes the behavior of a Web Service in terms of choreographed activities.

Figure 2.11 describes a part of an orchestration process of a customer. This process is derived from an internet purchase process that is entirely specified in appendix B.

```
<process name="internetPurchaseOrderCustomerProcess"
...
<sequence>
...
     <if>
         <condition>
            purchaseAcceptance == "true"
         </condition>
        <then>
            <sequence>
                <assign>
                    <copy>
                        <from>
                          <literal> "00:00" </literal>
                        </from>
                        <to variable=ClockX/>
                    </copy>
                </assign>
                <request partnerLink="Carrier"
                        portType="custosellPT"
                        operation="SendAcceptance"
                        variable="purchaseAcceptance">
                </request>
            </sequence>
         </then>
     </if>
...
</sequence>
</process>
```

Figure 2.11: Part of a Customer WS-BPEL process

CHAPTER 3

Model Checking applied to Real Time Systems and Probabilistic Systems.

Model checking can be applied to a wide range of systems, but are of special interest in the case of real-time systems, due to the peculiarities of these systems, in which failures may have catastrophic consequences. Two aspects should be considered when describing real-time systems, the first one, and in fact, the main important one, is related with time restrictions that described systems must fulfill, and the second one is related with probabilities, for instance, to measure the probability that a certain event has to occur, or a datum of a certain type to arrive.

Some different formalisms can be used to describe a concurrent system taking into account both factors (see [49, 93]). We particulary use timed automata and probabilistic transition systems for this purpose, and the UPPAAL tool that supports this formalism, as well as its probabilistic version, P_UPPAAL, which uses an integer semantic for time.

3.1 Introduction

Systems in which time aspects play a crucial role are known as Real Time Systems. These systems control the core of industrial, financial and governmental systems where the response time determines the grade of correctness, efficiency, user satisfaction, and other variables of quality. However, the set of systems where time really plays a critical role is that with **“strong time restrictions"**. And the meaning of “strong time restriction" is related to those systems where a delay in the execution of a task causes the system crash. This implies that it is necessary to check if the system really is free of faults. There are many verification techniques that cover the different phases of software life cycle. However, the most effective of them are centered in the first phases. Their effectiveness is based on the early fault managing that minimizes the development cost. This is because the discover of a fault in the last phases leads to double the cost of software, as it is necessary to back again to the beginning in order to solve the problem.

Another useful information in this kind of system is the probability of certain events to occur. In many cases, probabilities are related to systems failures, so it is very useful to know in advance that these situations may happen, and with which probability. As these failures are in many cases related to external factors, these are not always software failures, i.e., in some cases they are caused by natural diseases, weather, physics rules, etc. all of them uncontrollable. However, it is very useful to predict these events with a reasonable probability. Thus, we can see that there are systems where the combination of two factors “time and probability" determine the main characteristics. The verification techniques should not only control the time issues, they should also control the probability for certain events to occur.

In order to study probabilistic and real time systems, we use two different tools that perform model checking over these type of systems. The first tool to study is the UPPAAL tool, which checks real time systems, and the second is Probabilistic_UPPAAL (P_UPPAAL), which checks probabilistic systems.

UPPAAL is based on timed automata to capture the system behavior, and temporal logic as specification language to gather the system properties. This

model checker verifies the properties over the model in order to prove whether the properties are hold by the model or not.

As an extension of UPPAAL, **P_UPPAAL** uses probabilistic transition systems to capture the system behavior and a restrictive constraint language to gather the system properties. Like UPPAAL, this model checker verifies whether the system model holds the specified properties or not.

In the following sections we describe UPPAAL, P_UPPAAL, and the relationship between them.

3.2 UPPAAL - A tool for Automatic Verification of Real-Time Systems

UPPAAL is a tool suite for automatic verification of safety and bounded as well as unbounded liveness properties of real-time systems modeled as networks of timed automata [10, 11, 92, 97, 96]. The UPPAAL engine analyze properties of networks of Timed Automata using efficient constraint solving techniques. UPPAAL also supports diagnostic model-checking providing diagnostic information in the case that the verification of a particular real-time systems fails. The current version of UPPAAL is available at http://www.uppaal.com. The UPPAAL development started during the spring of 1995, and nowadays it is being extended with many features as distribution, guided, parameterized, cost-optimal, hierarchical (UML) or probability.

3.2.1 Real-Time Systems

UPPAAL uses *timed transition systems* (TTS for short) as the basic semantic model. The type of systems treated by UPPAAL is a particular class of timed transition systems that are syntactically described by *networks of timed automata* [91, 140].

A timed transition system is a labeled transition system with two types of labels: atomic actions and delay actions (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 7. *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 $V : S \rightarrow 2^P$ is a proposition assignment function.*

Note that the above definition is standard for labeled transition systems except that we have introduced a proposition assignment function V, which for each state $s \in S$ assigns a set of atomic propositions $V(s)$ that hold in s. Furthermore, we introduce a number of constraints on TTS, for instance, additivity and time determinism, which are defined as follows, respectively:

- *(Additivity)* $s \xrightarrow{\in(d_1+d_2)} s_1 \Rightarrow \exists s_2 \mid s \xrightarrow{\in(d_1)} s_2 \xrightarrow{\in(d_2)} s_1$
- *(Time Determinism)* $(s \xrightarrow{\in(d)} s_1 \wedge s \xrightarrow{\in(d)} s_2) \Rightarrow s_1 = s_2$

In order to study compositional problems we introduce a parallel composition between timed transition systems. Following [77] we suggest 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 let 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:

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

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 of timed transition systems.

3.2.2 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 [123] interpretation of parallel composition (proposed in [140]), allowing one-to-one communication and interleaving.

Networks of Timed Automata

By definition, a timed automaton is a standard finite-state automaton 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. UPPAAL extends the notion of timed automata to include integer variables, i.e. integer valued variables that may appear freely in general arithmetic expression used in guards as well as in assignment. Note that for backward reachability analysis, which we will define after, the variable assignment is restricted to any value of the form $ax + b$ where $a, b \in \mathbf{Z}$ and x is the variable being reassigned, whereas for forward reachability analysis there is not such restriction.

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

Definition 8. (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 the form: $x \sim n$ or $x - y \sim n$, for $x, y \in C$, $\sim \in \{\leq, \geq, =\}$and $n \in \mathbf{N}$. An atomic constraint over I is a constraint of the form: $i \sim n$, for $i \in I$, $\sim \in \{\leq, \geq, =\}$and $n \in \mathbf{Z}$.*

By $C_c(C)$ we will denote the set of all clock constraints over C, and by $C_i(I)$ we will denote the set of all integer constraints over I.

Definition 9. (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 $c \in (C_c(C) \bigcup C_i(I))$.*

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

Definition 10. (Assignments) *Let C be a set of real valued clocks and I a set of integer valued variables. A clock assignment over C is a tuple $\langle v, c\rangle$, where $v \in C$ and $c \in \mathbf{N}$. An integer assignment over I is a tuple $\langle w, d\rangle$ representing the assignment $w = d$, where $w \in I$ and $d \in \mathbf{Z}$.*

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

Definition 11. (Timed automata) *A timed automaton A over a finite set of actions Act, clocks$C$ and integer variables I is a tuple $\langle L, l_0, E, V\rangle$, where L is a finite set of nodes (control-nodes), l_0 is the initial node, $E \subseteq L \times \mathcal{B}(C, I) \times Act \times \mathcal{A}(C, I) \times L$ corresponds to the set of edges, and $V : L \to \mathcal{B}(C, I)$ assigns invariants to locations. For a brief notation, we will denote $l \xrightarrow{g,a,r} l'$ by the edge $\langle l, g, a, r, l'\rangle \in E$.*

Definition 12. (Synchronization Function) *Let $\mathcal{T} \subseteq Act \times Act$ be a function such that:*

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

Definition 13. (Parallel Composition) *Let A_1, A_2 be two timed automata. The parallel composition $(A_1|A_2)$ is a timed automaton $\langle L, l_0, E\rangle$, where $(l_1|l_2) \in L$ whenever $l_1 \in L_1$ and $l_2 \in L_2$, $l_0 = (l_{1,0}|l_{2,0})$, and the edges E are defined as follows:*

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

A **state** of a timed automaton A is a pair $\langle l, u\rangle | u \in V(l)$, where l is a node of 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 $\mathbf{Z}$, and $u \in V(l)$ means that u satisfies the invariant of the node l. We will use $g(u)$ 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 and clocks to 0.

The evolution of a timed automaton, from a state to another state can proceed by two types of transitions:

- Delay transition: $\langle l, u \rangle \xrightarrow{\in(d)} \langle l, u' \rangle$.
- Action transition: $\langle l, u \rangle \xrightarrow{g,a,r} \langle l', u' \rangle$.

Definition 14. (Delay transitions) *Let $\langle l, u \rangle$ and $\langle l', u' \rangle$ be two states of a timed automaton A, and let d be a positive real. Then*

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

Definition 15. (Action transition) *Let $\langle l, u \rangle$ and $\langle l', u' \rangle$ be two states of a timed automata A. Then*

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

Timed Automata in Uppaal

The Uppaal modelling language extends timed automata with the following additional features:

- *Templates* automata are defined with a set of parameters that can be of any type (e.g., `int`, `chan`). These parameters are substituted for a given argument in the process declaration.
- *Constants* are declared as `const name value`. Constants by definition cannot be modified and must have an integer value.

- *Bounded integer variables* are declared as `int[min,max]` name, where `min` and `max` are the lower and upper bound, respectively. Guards, invariants, and assignments may contain expressions ranging over bounded integer variables. The bounds are checked upon verification and violating a bound leads to an invalid state that is discarded (at run-time). If the bounds are omitted, the default range of -32768 to 32768 is used.

- *Binary synchronization* channels are declared as `chan c`. An edge labeled with `c!` synchronizes with another labeled `c?`. A synchronization pair is chosen non-deterministically if several combinations are enabled.

- *Broadcast channels* are declared as `broadcast chan c`. In a broadcast synchronization one sender c! can synchronize with an arbitrary number of receivers `c?`. Any receiver than can synchronize in the current state must do so. If there are no receivers, then the sender can still execute the `c!` action, i.e. broadcast sending is never blocking.

- *Urgent synchronization* channels are declared by prefixing the channel declaration with the keyword `urgent`. Delays must not occur if a synchronization transition on an urgent channel is enabled. Edges using urgent channels for synchronization cannot have time constraints, i.e., no clock guards.

- *Urgent locations* are semantically equivalent to adding an extra clock x, that is reset on all incoming edges, and having an invariant `x<=0` on the location. Hence, time is not allowed to pass when the system is in an urgent location.

- *Committed locations* are even more restrictive on the execution than urgent locations. A state is committed if any of the locations in the state is committed. A committed state cannot delay and the next transition must involve an outgoing edge of at least one of the committed locations.

- *Arrays* are allowed for clocks, channels, constants and integer variables. They are defined by appending a size to the variable name, e.g. `chan c[4]; clock a[2]; int[3,5] u[7];`.

- *Initializers* are used to initialize integer variables and arrays of integer variables. For instance, `int i := 2; or int i[3] := 1, 2, 3;`.

Expressions in Uppaal

Expressions in Uppaal range over clocks and integer variables. The BNF is given in Figure 3.1. Expressions are used with the following labels:

- *Guard* A guard is a particular expression satisfying the following conditions: it is side-effect free; it evaluates to a boolean; only clocks, integer variables, and constants are referenced (or arrays of these types); clocks and clockdifferences are only compared to integer expressions; guards over clocks are essentially conjunctions (disjunctions are allowed over integer conditions).

- *Synchronisation* A synchronisation label is either on the form `Expression!` or `Expression?` or is an empty label. The expression must be side-effect free, evaluate to a channel, and only refer to integers, constants and channels.

- Assignment An assignment label is a comma separated list of expressions with a side-effect; expressions must only refer to clocks, integer variables, and constants and only assign integer values to clocks.

- Invariant An invariant is an expression that satisfies the following conditions: it is side-effect free; only clock, integer variables, and constants are referenced; it is a conjunction of conditions of the form `x<e` or `x<=e` where x is a clock reference and e evaluates to an integer.

3.2.3 Verifying Real-time Systems in UPPAAL

As mentioned in the introduction, the increasing use of systems dealing with time requirements makes it necessary to develop methods to check the system behavior correctness.

As pointed out in [140, 65], the practical goal of verification of real-time systems, is to verify simple safety properties such as “can we guarantee that

Expression	→	ID | NAT
		| Expression '[' Expression ']'
		| '(' Expression ')'
		| Expression '++' | '++' Expression
		| Expression '– –' | '– –' Expression
		| Expression AssignOp Expression
		| UnaryOp Expression
		| Expression BinaryOp Expression
		| Expression '?' Expression ':' Expression
		| Expression '.' ID
UnaryOp	→	'–' | '!' | 'not'
BinaryOp	→	'<' | '<=' | '==' | '!=' | '>=' | '>'
		| '+' | '–' | '*' | '/' | '%' | '&'
		| '|' | '^' | '<<' | '>>' | '&&' | '||'
		| '<?' | '>?' | 'and' | 'or'| 'imply'
AssignOp	→	':=' | '+=' | '–=' | '*=' | '/=' | '%='
		| '|=' | '&=' | '^=' | '<<=' | '>>='

Figure 3.1: Syntaxis of expressions in BNF

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 now considered are infinite-state because of the clocks are real-valued.

Specifying Properties in TCTL Style

The language that UPPAAL uses to perform the verification is a subset of timed computation tree logic (TCTL,[ACD93]), where atomic expressions are location names, variables and clocks gathered from the modeled system. The properties are defined using local properties that are either true or false by depending on a specific configuration.

Definition 16. (Local Property) *Given an UPPAAL model $\vec{A}$. A formula φ is a local property iff it is formed according to the following syntactical rules:*

$\varphi ::=$	$deadlock$	
$\mid$	$A.l$	$for\ A \in \vec{A}\ and\ l \in L_A$
$\mid$	$x \bowtie c$	$for\ x \in Clocks, \bowtie \in \{<, <=, ==, >=, >\}, c \in \mathbb{Z}$
$\mid$	$x - y \bowtie c$	$for\ x, y \in Clocks, \bowtie \in \{<, <=, ==, >=, >\}, c \in \mathbb{Z}$
$\mid$	$a \bowtie b$	$for\ a, b \in Vars \bigcup \mathbb{Z}, \bowtie \in \{<, <=, !=, ==, >=, >\}$
$\mid$	(φ_1)	$for\ \varphi_1\ a\ local\ property$
$\mid$	$not\ \varphi_1$	$for\ \varphi_1\ a\ local\ property$
$\mid$	$\varphi_1\ or\ \varphi_2$	$for\ \varphi_1, \varphi_2\ logical\ properties\ (logical\ OR)$
$\mid$	$\varphi_1\ and\ \varphi_2$	$for\ \varphi_1, \varphi_2\ logical\ properties\ (logical\ AND)$
$\mid$	$\varphi_1\ imply\ \varphi_2$	$for\ \varphi_1, \varphi_2\ logical\ properties\ (logical\ implication)$

Thus, the truth value of a local property can be evaluated in a local configuration, as follows:

Definition 17. (Correctness of a Local Property) *A local property φ is correct in a local configuration $\mathbf{s} = \langle l, u \rangle | u \in V(l)$, denoted by $\mathbf{s} \vDash_{loc} \varphi$, iff it is correct according to the following definitions:*

$\mathbf{s} \vDash_{loc} deadlock$	*iff*	*no delay or action steps are enabled in* $\mathbf{s}$
$\mathbf{s} \vDash_{loc} A.l$	*iff*	$l = l_i\ in\ \vec{l}\ for\ A = A_i\ in\ \vec{A}$
$\mathbf{s} \vDash_{loc} x \bowtie c$	*iff*	$u(s) \bowtie c, \bowtie \in \{<, <=, ==, >=, >\}$
$\mathbf{s} \vDash_{loc} x - y \bowtie c$	*iff*	$u(x) - u(y) \bowtie c, \bowtie \in \{<, <=, ==, >=, >\}$ *and* $x, y \in C$ *(Clocks)*
$\mathbf{s} \vDash_{loc} a \bowtie b$	*iff*	$u(a) \bowtie u(b), \bowtie \in \{<, <=, !=, ==, >=, >\}$ *and* $a, b \in I$ *(Vars)*
$\mathbf{s} \vDash_{loc} (\varphi_1)$	*iff*	$\mathbf{s} \vDash_{loc} \varphi_1$
$\mathbf{s} \vDash_{loc} not\ \varphi_1$	*iff*	$\neg(\mathbf{s} \vDash_{loc} \varphi_1)$
$\mathbf{s} \vDash_{loc} \varphi_1\ or\ \varphi_2$	*iff*	$\mathbf{s} \vDash_{loc} \varphi_1\ or\ \mathbf{s} \vDash_{loc} \varphi_2$
$\mathbf{s} \vDash_{loc} \varphi_1\ and\ \varphi_2$	*iff*	$\mathbf{s} \vDash_{loc} \varphi_1\ and\ \mathbf{s} \vDash_{loc} \varphi_2$
$\mathbf{s} \vDash_{loc} \varphi_1\ imply\ \varphi_2$	*iff*	$\neg(\mathbf{s} \vDash_{loc} \varphi_1)\ or\ \mathbf{s} \vDash_{loc} \varphi_2$

Where φ_1 and φ_2 stand for local properties.

This notion of locality must not be confused with locality in the sense of "locality to a process". The temporal logic of UPPAAL language let the user define local variables and clocks of a process by using the syntax, $P.var$, which represents a variable, var, local to the process, P.

$E <> \varphi$	reachability of φ
$A[\ ]\ \varphi$	safety (invariantly φ)
$E[\ ]\ \varphi$	possibly always φ
$A <> \varphi$	inevitably φ
$\varphi - - > \psi$	unbounded response
	(corresponds to $A[\ ]\ (\varphi \Rightarrow A <> \psi)$)

φ, ψ: local properties

Figure 3.2: The five different kinds of temporal properties for UPPAAL.

In definitions 16 and 17 we have expressed the syntax of the temporal logic that UPPAAL uses, and figure 3.2.3 shows the five different kinds of temporal properties that UPPAAL can verify. Let us see the definition of three of them, the remaining two clases are defined easily from them.

Definition 18. (Temporal Properties) *Let $\langle l, u\rangle$ be a control state of a Timed Automata and let φ and ψ be local properties. The correctness of temporal properties is defined for the classes $A[\ ]$, $A <>$ and $- - >$ as follows:*

$$
\begin{array}{lll}
\langle l,u\rangle \vDash A[\ ]\ \varphi & \textit{iff} & \forall \langle l',u'\rangle .\ \langle l,u\rangle \rightarrow^* \langle l',u'\rangle \Rightarrow \langle l',u'\rangle \vDash_{loc} \varphi \\
\langle l,u\rangle \vDash A <> \varphi & \textit{iff} & \exists \langle l',u'\rangle .\ \langle l,u\rangle \rightarrow^* \langle l',u'\rangle \Rightarrow \langle l',u'\rangle \vDash_{loc} \varphi \\
\langle l,u\rangle \vDash \varphi - - > \psi & \textit{iff} & \textit{if } \forall \langle l'_1,u'_1\rangle .\ \langle l,u\rangle \rightarrow^* \langle l'_1,u'_1\rangle \Rightarrow \langle l'_1,u'_1\rangle \vDash_{loc} \varphi \textit{ then} \\
 & & \exists \langle l'_2,u'_2\rangle .\ \langle l'_1,u'_1\rangle \rightarrow^* \langle l'_2,u'_2\rangle \Rightarrow \langle l'_2,u'_2\rangle \vDash_{loc} \psi
\end{array}
$$

The other temporal property classes are dual to $A[\ ]$ and $A <>$, and they are defined as follows:

$$
\begin{array}{lll}
\langle l,u\rangle \vDash E <> \varphi & \textit{iff} & \neg (M \vDash_{loc} A[\ ]\ \textit{not}(\varphi)) \\
\langle l,u\rangle \vDash E[\ ]\ \varphi & \textit{iff} & \neg (M \vDash_{loc} A <>\ \textit{not}(\varphi))
\end{array}
$$

Symbolic Model-Checking

The region-graph technique pointed out in [65, 140] 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-independent and the number of such regions depends highly on the constants used in the clocks constraints of an

automaton, 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 automaton 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 automaton 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 of 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 into account of particular property, the number of partitions is in practice considerably smaller compared with the region-technique.

Reachability Analysis by Constraint Solving

Adopting the methodology developed in [140], we start by describing a simple reachability problem for timed automata. A generalized version of the problem will be given later.

Definition 19. (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 $\langle l_0, u_0 \rangle \rightarrow^* \langle l_f, u_f \rangle$ 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 20. (General Reachability) *Let l_0, l_f be nodes of a timed automaton 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 now describe an algorithm for solving the general reachability problem, using constraint solving techniques.

Let A be the timed automaton 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 solution set of a constraint system r (i.e. the set of all solutions of r) can be seen as a convex polytope in the n-dimensional space. It can be seen in Figure 3.3.

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 of this form can easily be represented by matrices.

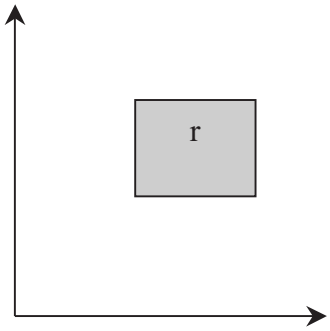


Figure 3.3: A solution region r

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 constraint solving. The guards and resets on the transitions and delays of the states are the things that affect the constraints and motivate the operations introduced. The direction of the reachability analysis, forwards or backwards, will need different operations on constraint system.

Definition 21. (Delay operations)

1. *The weakest precondition of a constraint system r is defined as follows:*

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

2. *The* strongest postcondition *of a constraint system r is defined as follows:*

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

These operations are needed because the automata, perform several 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 forwards for $\langle l_f, U_f \rangle$. In this case the strongest postcondition is used. If we do backward reachability analysis we

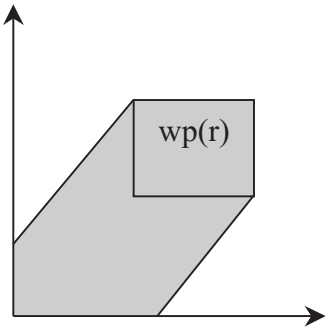


Figure 3.4: The weakest precondition operation over a Constraint System r

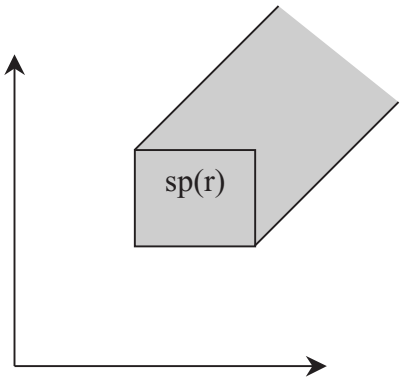


Figure 3.5: The strongest precondition operation over a Constraint System r

search backwards 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 Figures 3.4 and 3.5 respectively.

Definition 22. (Guard conjunction) *If r is a constraint system and g is a set of guards the time region after the guard conjunction is the intersection of the solution sets: $g \cap r$.*

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

Definition 23. (Resetting clocks) *Let r be a constraint system and k the clock*

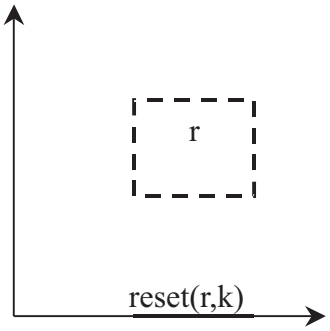


Figure 3.6: The reset operation over a Constraint System r

to be reset.

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

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

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

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

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

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

The reset operation is used in forward reachability analysis to handle the resets that may be performed when the system performs a transition. In backward reachability analysis the free and check operations are used to handle the resets on the transitions. It is straightforward 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 Figures 3.6 and 3.7, respectively.

The implementation of the algorithm and operations are much easier if the constraint systems are of a special form, called closed.

Definition 24. *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:*

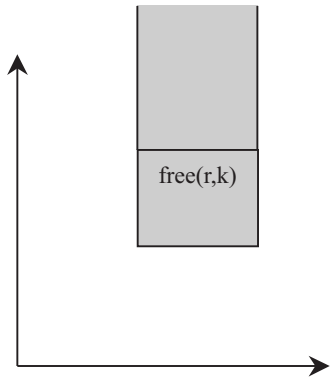


Figure 3.7: The free operation over a Constraint System r

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

Constraint systems are closed under the operations defined above.

3.2.4 Algorithms

We now present a reachability analysis algorithm using constraint solving. The algorithm checks whether 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 contains the states not yet explored and the passed buffer holds the states explored so far.

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. (Forward Reachability Analysis)

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

2. *Check if* $l_i = l_f \wedge U_i \subseteq U_f$. *In this case, return the answer yes.*

3. *if $l_i = l_j \wedge U_i \subseteq U_j$, for some $\langle l_j, U_j \rangle$ 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 into account only the actions . Let g_k be the set of guards on the performed transition an a_k the set of resets.*

5. *We now take $U_k = reset(sp(U_i) \cap g_k, a_k)$. 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.*

Backward Reachability Analysis

To simplify the algorithm description below we define an operation *inv_reset*, which consists of operations defined above.

Definition 25. *Let "r" be a constraint system and "a" a set of resets. We define*

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

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

Algorithm 2. (Backward Reachability Analysis)

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

2. *Check if $l_i = l_0 \wedge U_i \subseteq U_0$. In this case, return the answer yes.*

3. *if $l_i = l_j \wedge U_i \subseteq U_j$, for some $\langle l_j, U_j \rangle$ 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 lead to l_i in one step regardless of the assignments, taking into account only actions. Let g_k be the set of guards on the performed transition an a_k the set of resets.*

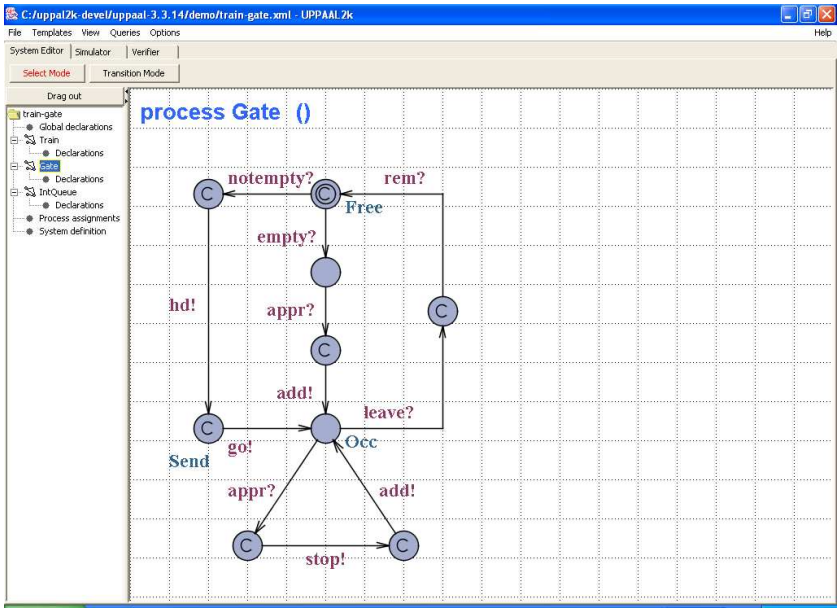


Figure 3.8: Overview of UPPAAL

5. *We take* $U_k = inv\_reset(wp(U_i) \cap g_k, a_k)$. *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.*

3.2.5 An Overview of the UPPAAL GUI

The UPPAAL's [11, 92, 96] main window (Figure 3.8) consists of two parts: the option's menu and the panel of tabs. The option's menu is described in the integrated help, which is accessible through the help option in the menu. This option also describes the UPPAAL functionality in detail. The tab panel is divided into three tab panels and these tab panels give access to three work areas of UPPAAL, which are the editor, the simulator and the verifier. Figure 3.8 shows the editor view. The idea is to define a bunch of templates (like in C++) that are instantiated in order to get a complete system. The templates can define symbolic variables and constants as parameters.

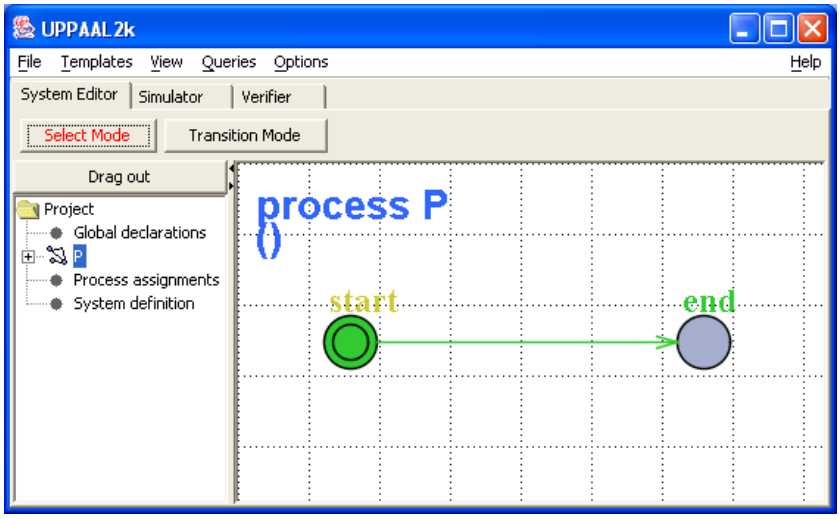


Figure 3.9: A first example

In order to get a first contact with UPPAAL we make a double click in the drawing area to get a location, if you repeat this, you have two. By a double click on these locations you can rename them as “start" and “end". Click on the Transition Mode, click on the start location and on the end location. Then, you have your first automata ready, as depicted in Figure 3.9. 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 3.10 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 you have the variables and on the right the system itself.

The last tab is the Verifier, which is shown in Figure 3.11. It is divided into 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 into 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

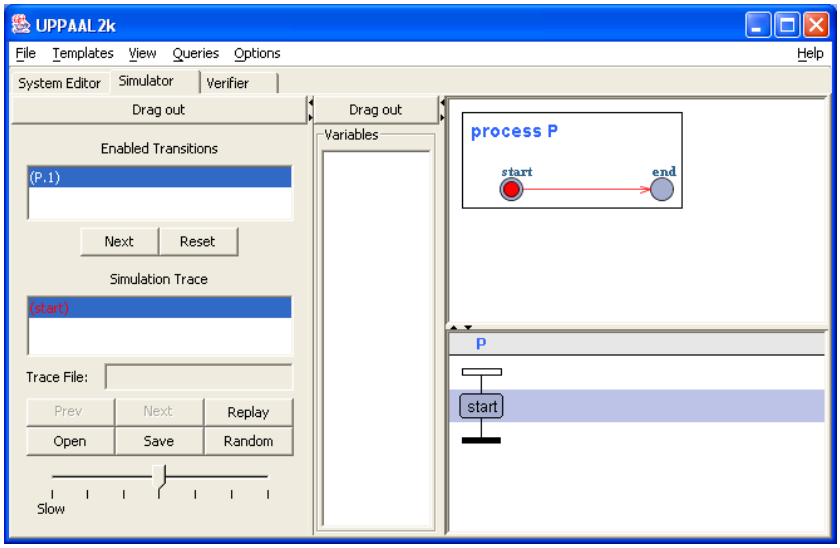


Figure 3.10: A snapshot of the simulator

some buttons which allows us to check the properties, change the mode between the property view or comment view and remove and insert new properties. All of this can be seen in Figure 3.11.

The reader can find a small tutorial on UPPAAL at the URL:
http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf
and several case studies in [1, 12, 16, 17, 118, 67, 68, 80, 94, 95, 101, 103].

3.3 The P_UPPAAL tool for probabilistic systems

As mentioned above, the P_UPPAAL tool (Probabilistic_UPPAAL) [47, 49] deals with probabilistic systems. This kind of system is characterized by the appearance of non deterministic information and actions that make the system unstable, although it is possible to compute and study the probability of occurring. This section describes how P_UPPAAL tool models these systems and how this tool can be used to check probabilistic properties.

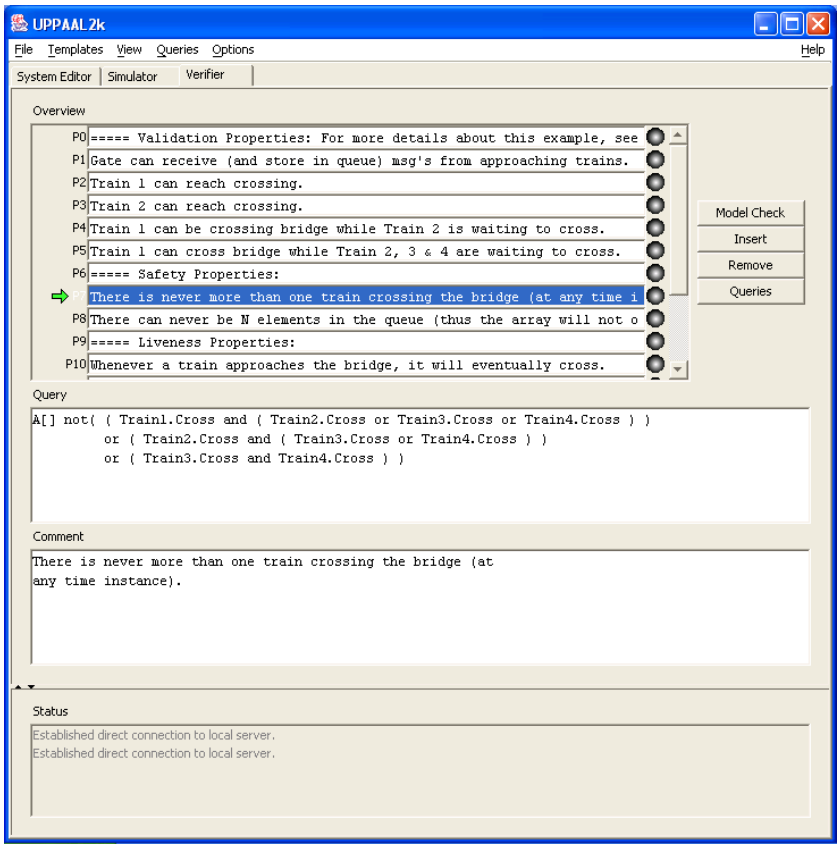


Figure 3.11: A snapshot of the verifier

3.3.1 Introduction

With P_UPPAAL we can model a timed-probabilistic system, but we can also simulate and analyze the system behavior. Simulations are very useful in these tools in order to detect some disfunctions in the normal conditions of the system. They are usually used as a first step for the validation of a system. The modeling and simulation activities are led through a graphical interface and the verification process is accomplished by RAPTURE [42, 43, 79]. This tool, P_UPPAAL, introduces the probabilistic behaviors to the UPPAAL models. To add these behaviors, the tool imports models from UPPAAL and translates them into a graphical model. Then, 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 [119] 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", probabilistic properties of the type “a desired state will be reached with probability of at least 99%", and properties that combine both “a desired state will be reached within 105 seconds with probability of at least 99%".

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

Notice that P_UPPAAL does not require UPPAAL to work, it allows modeling the system using the provided editor and after that, it exports the system

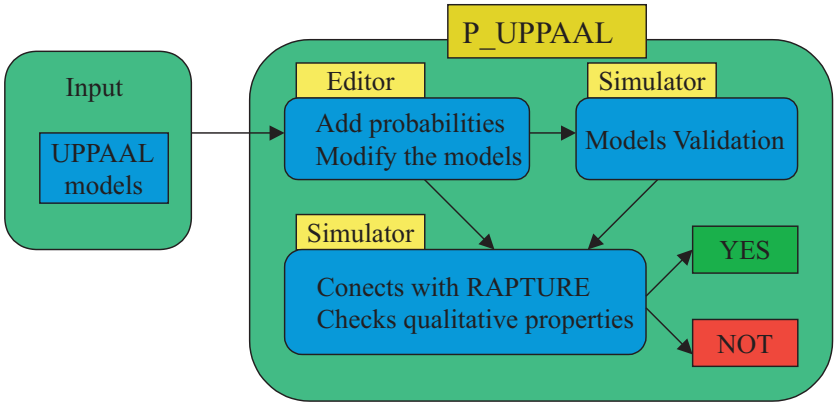


Figure 3.12: Architecture of the P_UPPAAL tool

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.

3.3.2 P_UPPAAL Architecture and Functionality

The architecture of P_UPPAAL, shown in Figure 3.12, is divided into three main parts, the Editor, the Simulator and the Verifier.

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

The Simulator allows us to analyze and validate the most relevant behaviors of the model by following its traces. Thus, we can analyze the traces and then decide whether the model holds the behavior expected or not. The simulator does not use the probabilities, because the trace generator is lead by the verifiers. However, the model is inherently non-deterministic.

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

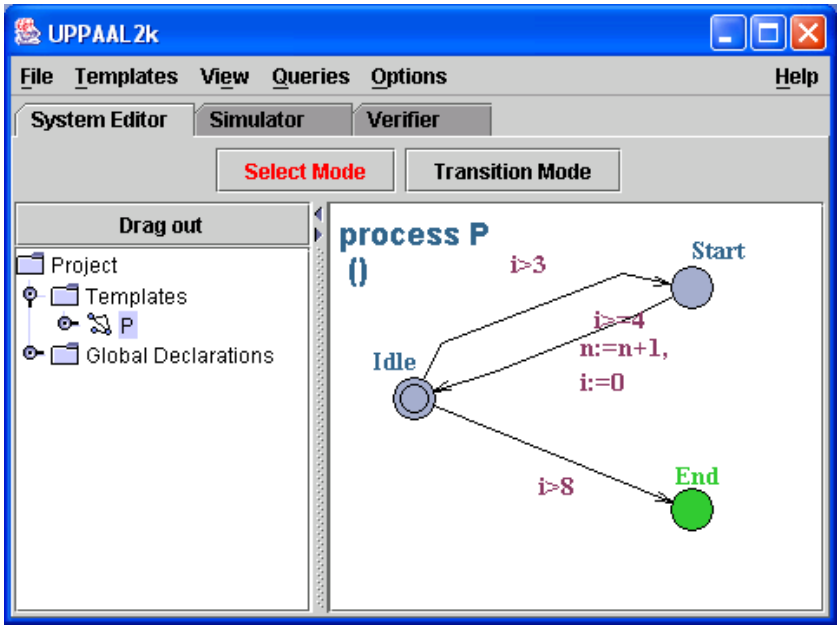


Figure 3.13: An example of UPPAAL model.

Importing Models from UPPAAL

The import process is based on the following idea: “The UPPAAL and P_UPPAAL models are derived from Labeled Transitions Systems". Thus, we just import from the UPPAAL models the states, the transitions, the variables and channels, the synchronization, and the guards. Therefore, we import the complete model with the exception of the timing characteristics (invariants and clocks).

Figures 3.13 and 3.14, show a simple example of the original model from UPPAAL and the imported model in P_UPPAAL, respectively. As you can see in Figure 3.14, the import model does not have the clock i which could be seen at the UPPAAL model in Figure 3.13.

We can find the *import option* at the P_UPPAAL menu, which opens an open dialog to search the UPPAAL model for the import activity.

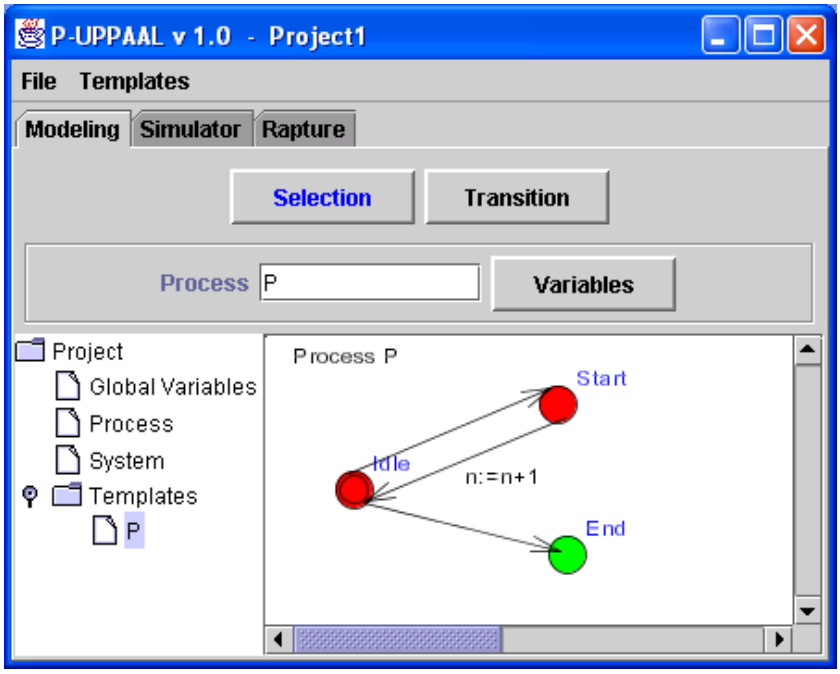


Figure 3.14: P_UPPAAL imported model from UPPAAL.

Modeling Systems

As mentioned above, P_UPPAAL consists of three tabs: the Editor tab, the Simulator tab and the verifier tab, which uses the RAPTURE tool. The first tab, the Editor, is a graphical interface that allows us to describe and draw the probabilistic transition system.

The Editor is divided into 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 the **variables** section, you can define integral variables and channels that provide the synchronization between the processes. In the **process** section, you can define the process from the different templates defined before. Finally, in the **system** section, 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 ways of working, the selection mode and the transition mode. *The selection mode* allows us to create or to remove states, as well as the edition of the state and transition properties. Thus, we can edit the name of the states and we can also establish or update the initial state. Furthermore, we can edit the transition properties by adding guards, assignments, synchronizations (with other processes through the channels) and probabilities. On the other hand, *the transition mode* allows us to depict the relations among the states, however, we must change to the selection mode to add properties to the transitions (as seen before).

Figure 3.15 depicts the tab Editor, with a simple example. This model just consists of one process, which also consists of three states: Idle, Start and End. The probability to reach the state End is just *0.1*, whereas the probability to reach the state Start is *0.9*. This example has been derived from the model in Figure 3.13, which has been modified with probabilities in transitions and some other changes.

System Simulator

The P_UPPAAL tab **Simulator** provides a way to validate whether the model captures appropriately the main characteristics of a given system or not. We can check if the model is correct, i.e., if the model holds the most representative design requirements that we have collected previously. The simulator is a path generator that allows verifiers to navigate through the model by executing the different transitions of the model; i.e, the verifier generates paths by executing transitions. This transition selection mode implies that the selection is deterministic. Thus, the probabilistic information is not taken into account in this simulation process. However, this information will be used in the next phase by the verifier.

To check the model syntax, P_UPPAAL uses a parser written in JAVA. The simulation algorithm used by P_UPPAAL is:

1. Checks the guards.

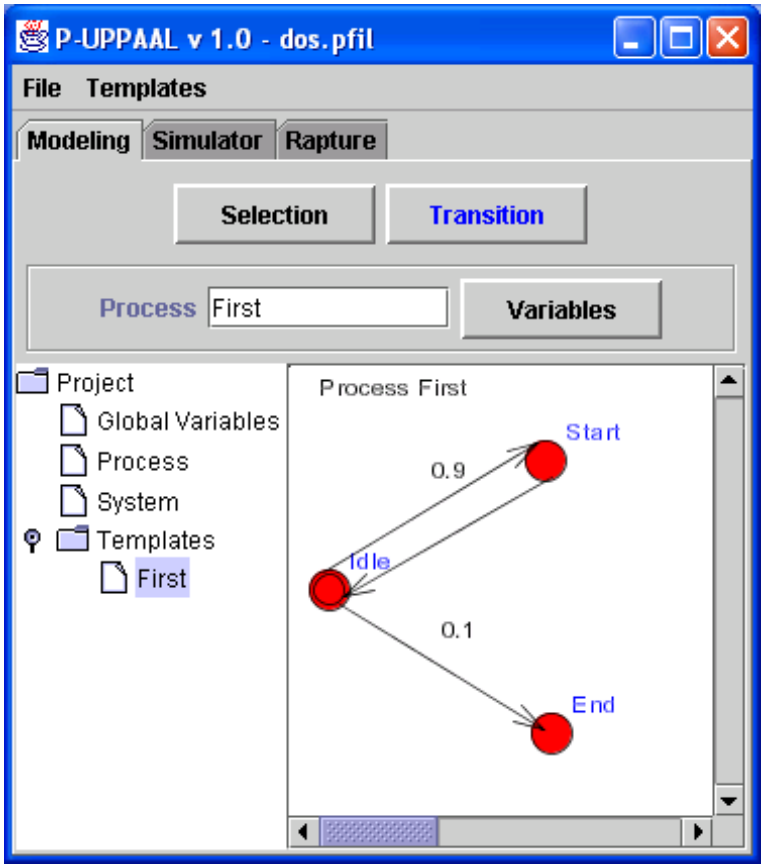


Figure 3.15: P_UPPAAL example.

2. Store the enabled transitions; if there are no enabled transitions then the simulation ends.

3. Choose the transition or transitions to be executed (in case of synchronizations two transitions must be executed at the same time).

4. Execute transition, which requires the following actions:

   - Make the assignments.
   - Make the synchronization.

5. Return to the beginning.

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

Before the verification process starts, we must save the system model. After that, you can start the simulation. The first step is to choose the transition to run, this is made through the “Enabled transition" panel. Then, you push “Next button" and the generated trace is stored in the trace panel.

Figure 3.16 shows the simulator tab, where a simulation process is depicted for the system of Figure 3.15.

The Verification Process

The verification process is accomplished through the RAPTURE tool. To connect with RAPTURE, P_UPPAAL provides the RAPTURE tab. This tab is divided into two parts. At the top, we can establish the initial and final state, and the probability to reach the final state from the initial state and, at the bottom, you can see the RAPTURE output.

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

Figure 3.17 depicts the RAPTURE verifier tab running.

The next subsection treats the RAPTURE tool into detail.

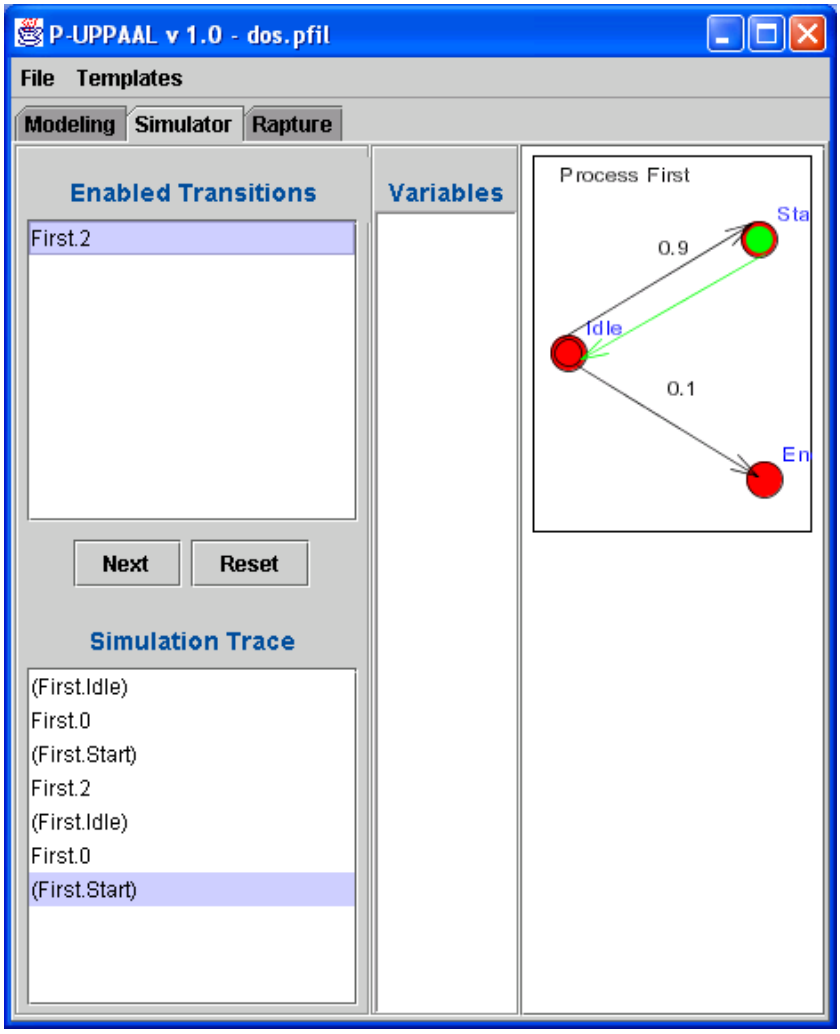


Figure 3.16: A simulation example.

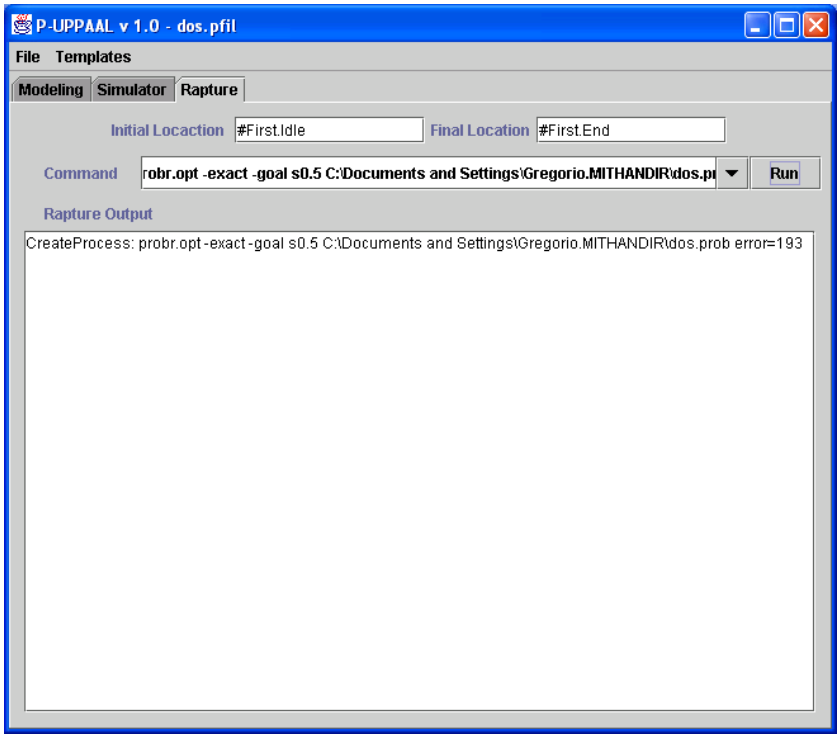


Figure 3.17: System verification.

3.3.3 RAPTURE - The Probabilistic Verification Engine

The context systems of RAPTURE are described in terms of Markov decision process [119], also called probabilistic transition systems (PTS) or probabilistic automata. This model allows us 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 [43]. 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 uses Bdds and Mtbdds (or Adds) [43, 42, 7].

Probabilistic Transitions Systems

As it is pointed out in [43, 42], 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. [82, 127]), and it is also known as Markov decision processes [119].

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

Definition 26. (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 system (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 $s \nrightarrow$. We let $sup(\pi) = \{s' | \pi(s') > 0\}$. We call s a *sink* state if $s \nrightarrow$.

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 $last(\sigma) \nrightarrow$, or an infinite sequence. We denote by $s - paths(T)$ and $f - paths(T)$ the sets of simple paths and full-paths 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-field 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 27. (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.

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 it is clear from the context.

Definition 28. (Extremum Probabilities) *The minimum 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*

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

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

We talk of an extremum probability to refer to either the minimum or the maximum probability.

We use the shorthand $\mathrm{P}^{inf}(s)$ and $\mathrm{P}^{sup}(s)$ for $\mathrm{P}_{T,s_0}^{inf}(s = s_0, \phi_f)$ and $\mathrm{P}_{T,s_0}^{sup}(s = s_0, \phi_f)$ 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 28 define extremum probabilities, but do not provide an effective way of computing them. However, it is well known [8, 13] that P^{inf} and P^{sup} can be characterized as the least fixpoints of operators $\mathrm{F}^{inf}, \mathrm{F}^{sup} = (S \rightarrow [0, 1]) \rightarrow (S \rightarrow [0, 1]))$ 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

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

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 pre-computation 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 minimum (resp. maximum) 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 [45] on a boolean abstraction of the system.

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

maximize P^{inf} under the constraints

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

minimize P^{sup} under the constraints

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

Simulations and Partitioning

Probabilistic simulation [127, 81] is central to state the correctness of the abstraction technique shown 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 29. (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 $s \rightarrow \pi$, there exist ρ such that $t \rightarrow \rho$ 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 an equivalence relation. The simulation $\preceq_C$ provides a sufficient condition for preservation of extremum probabilities, as the following theorem states:

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 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 30. (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 that $s_0 \in A \in \mathcal{A}$.

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 proposition 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 $\mathcal{B} \leq \mathcal{A} \leq \mathcal{C}$,*

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

RAPTURE Architecture Implementation

The architecture of RAPTURE is the following (see Figure 3.18):

1. *The frontend* parses the input language, which specifies both the system to be analyzed, the property and possibly the components (processes and variables) not abstracted in the initial 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. If it allows to prove or disprove the property, the verdict is emitted.

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

As stated before, RAPTURE uses linear programming 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 are 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 desired.

3.3.4 An example: Communication Through Unreliable media

In order to illustrate these two tools, P_UPPAAL and RAPTURE, we will model, validate and verify a simple example with them. This example consists of a client and a server which communicate through two unreliable channels (a channel for each way). The client sends a sequence of M different requests to the server. The server processes the requests and sends acknowledgements of them to the client. If there is a failure (we suppose that the client can detect it), the client can resend the request twice. On completion, the sender emits the message CORR, otherwise the message ERR will be emitted.

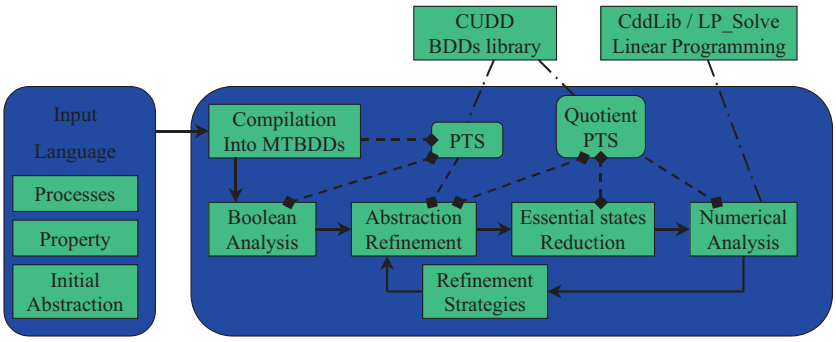


Figure 3.18: Architecture of the RAPTURE tool

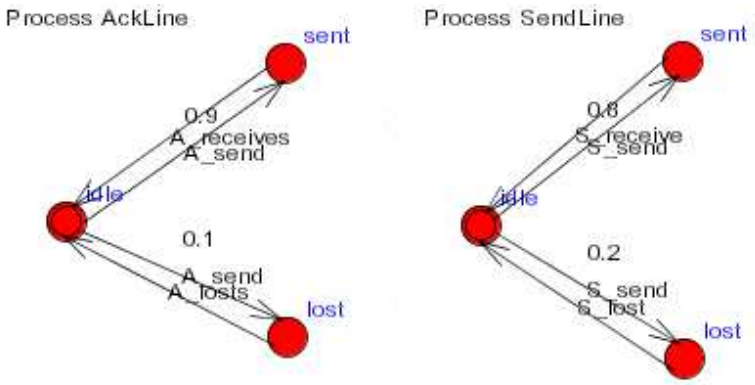


Figure 3.19: The two communication media: AckLine & SendLine

Global Declarations

We first declare the set of channels:

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

The Two Communication Media

We define the SendLine and AckLine processes(see Figure 3.19). The field synchronizations allows us to define the set of channels on which the process synchronize.

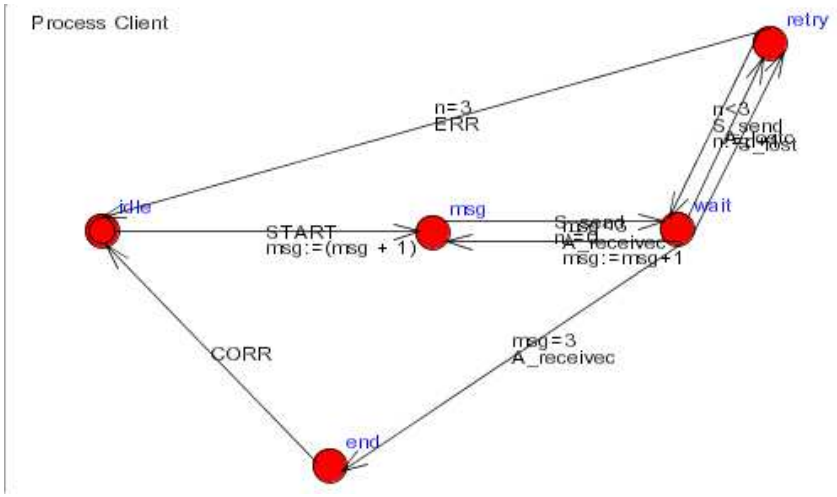


Figure 3.20: Client process

The Client Process

The more complex process is the Client process that you can see in Figure 3.20. This process owns two local variables. Their type is uint(2), which means: 2-bits unsigned integers.

The Server Process

Lastly, we have the (dummy) Server process that is shown in the Figure 3.21.

The Simulation

Figure 3.22 depicts the simulation process of the system, which allows us to validate the system model.

The Property and the Verification

In order to check the probability of: “one message has been correctly sent", is greater than “50%". We add the process sender into the system (see Figure 3.23). Furthermore we must set the “initial" field to “#Sender.idle” and the “final" field to “#Sender.end” in order to specify this property.

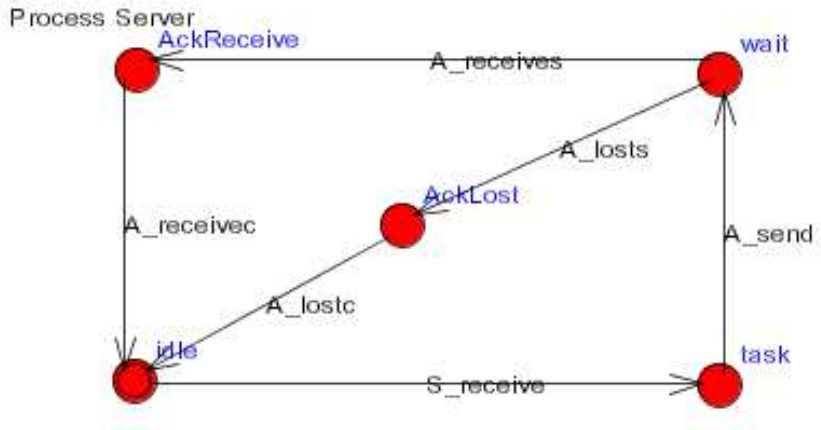


Figure 3.21: Server process

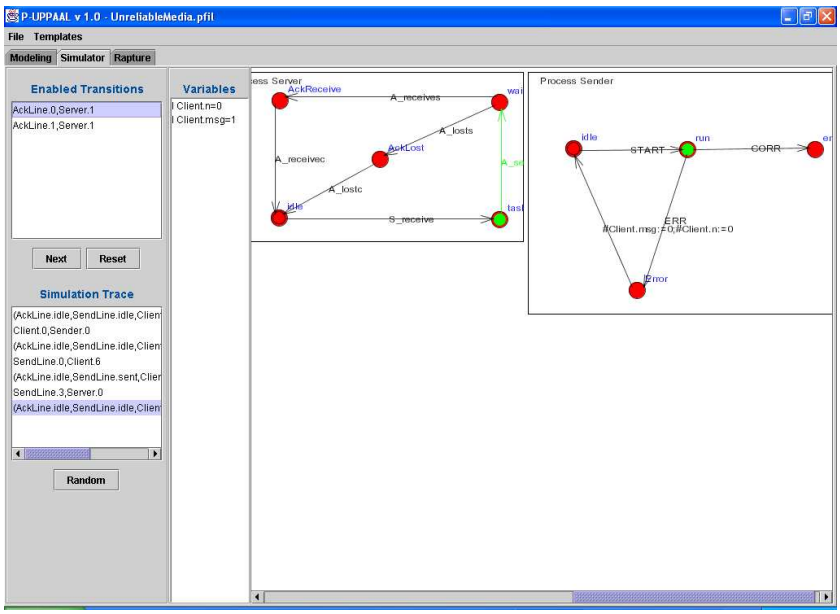


Figure 3.22: System simulation

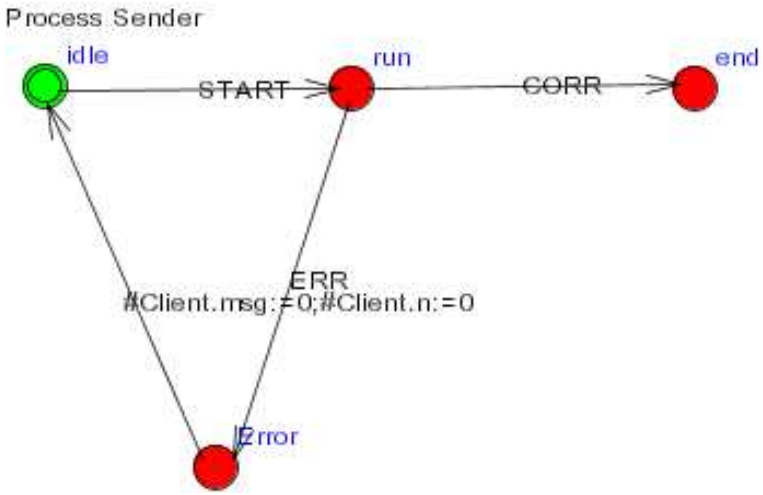


Figure 3.23: Sender process

The output provided by RAPTURE can be seen in Figures 3.24 and 3.25. The property is false, because the $P^{sup} \leq 0.5$ on the stabilized partition of the system. As every member of this partition is a bisimulation equivalence class, we get that $P^{sup} = 0.975639511753$ on the concrete system.

3.4 From UPPAAL to P_UPPAAL

We can translate timed automata from UPPAAL to Probabilistic Transition Systems for RAPTURE without losing the temporal references.

3.4.1 Translation from Timed Automata to Probabilistic Transition System

A timed automata, by definition, is a standard finite-state automaton extended with a finite collection of real valued clocks. Clocks are assumed to proceed at the same rate and their values may be compared with natural numbers or reset to 0. It is possible to extend 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

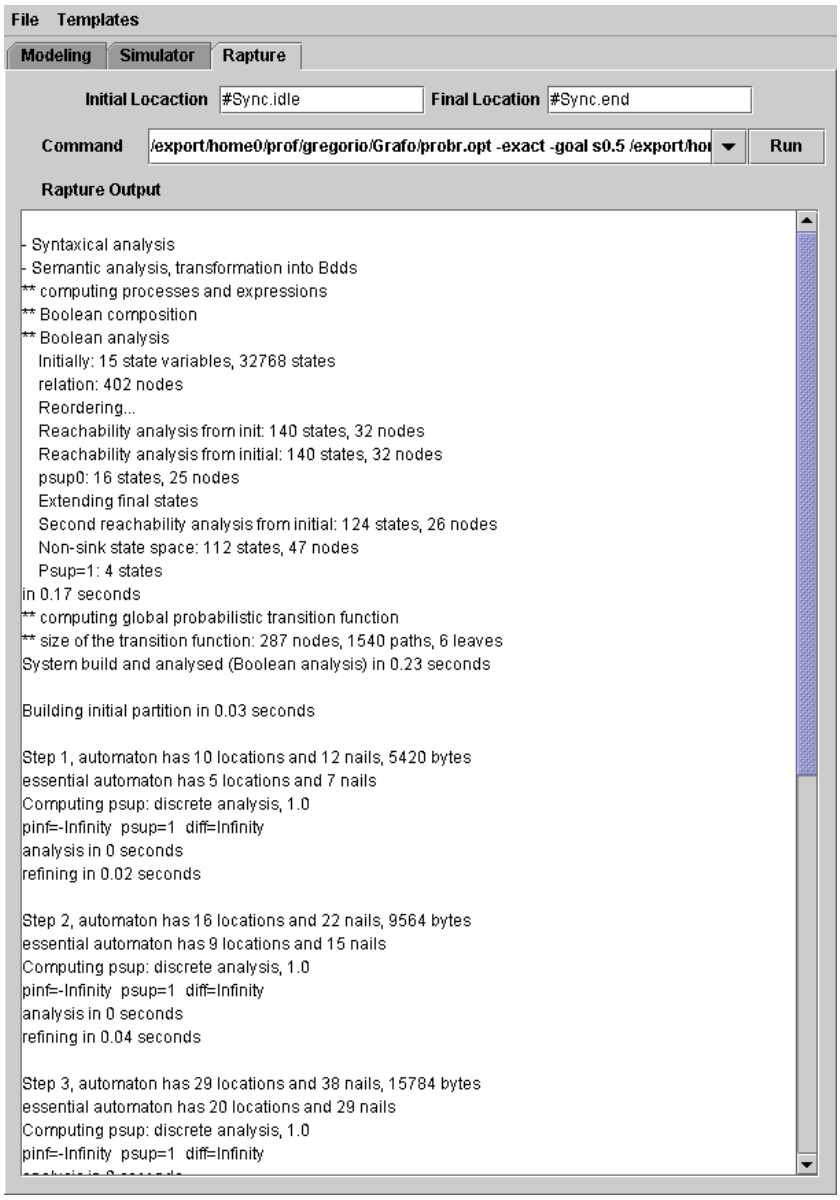


Figure 3.24: The verifier verdict

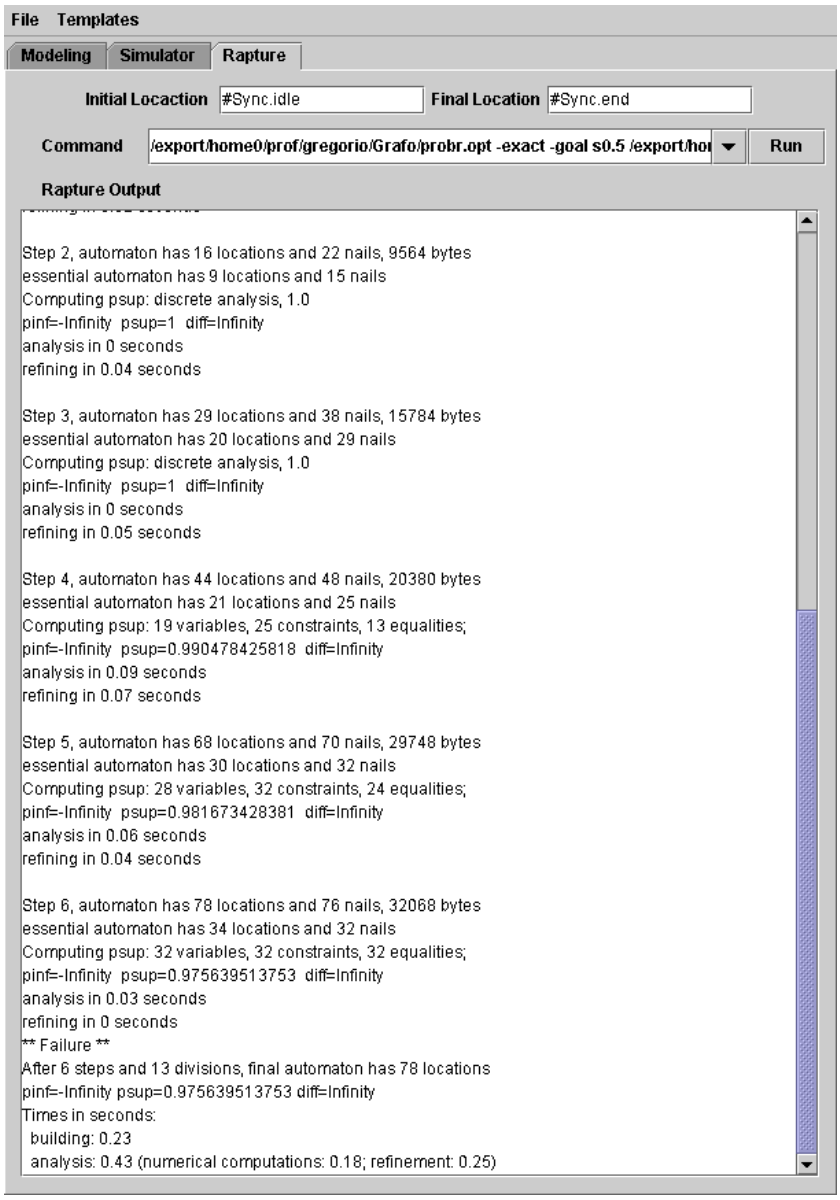


Figure 3.25: The verifier verdict

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

3.4.2 Keeping Temporal References

As we have seen above, the main difference between both models are the clocks and probabilistic extensions over the well-known transition system. Thus, in order to translate from Networks of Timed Automata into Probabilistic Transition Systems, it is necessary to remove the clocks from UPPAAL models, as it is pointed out in [48]. But in this case, it is not possible to check properties consisting of probabilistic and temporal references, because PTS cannot deal with clocks.

A possible solution is to translate the clocks into common elements of both models, integer variables. Thus, the clocks could be considered as integer variables, and then, we could reset or modify these clocks. Furthermore, we could use them in guards and in invariants, as we did before in the automata.

From the syntactic point of view this is correct, but furthermore we must consider the semantic point of view, i.e. how the automaton may evolve. As we have seen in definitions 14 and 15, the system can evolve either using a delay transition or an action transition. There are no problems when the system evolves by performing action transitions, although, we must check whether the guards use clocks or not in order to translate them into integer variables. But, if the system evolves by making a delay transition, the clocks cannot evolve, i.e. no time is elapsed, owing to the fact that they are now integer variables.

Therefore, we must consider a technique that allows the system to simulate the passage of time. An approach to deal with this problem is to add a loop transition in each node to increase the clocks. These new transitions do not have any guards or synchronizations. But, each loop has an assignment that increases all clocks, regardless they are global or local to one automaton.

The technique used in the translation is an “Integer Semantic", which is valid as a basic model to be extended with probabilities. The obtained output from the translation process will be the input for the probabilistic tool. Once specifiers have added the probabilistic extensions, verifiers can specify and check properties with probabilistic and temporal references.

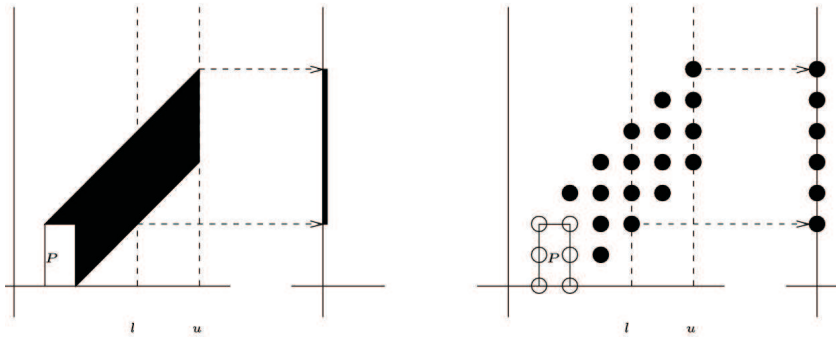


Figure 3.26: Constraint Solving of a constraint system P for the dense and discrete timed automata.

Thus, we have changed the real clocks for integer clocks or digital clocks [72]. This change also implies a change from the semantic view point, because, we have changed from a semantic where the time deals with “dense time" to a semantic that works with “discrete time", i.e., from a “Dense Semantic" to a “Discrete Semantic" (see [120, 20] for further information about dense and discrete semantics). The differences, which this change results in, are that not all the behaviors captured by a dense semantic can be captured with the discrete semantic. We can see the differences in the example of Figure 3.26, where the constraint solving of a constraint system for dense and discrete time automata is different for each different semantic.

Despite of the two semantics are not equivalent, there are cases where the discrete semantic behaves as the dense semantic. Figure 3.27 shows an example of a probabilistic timed automata where the integer semantic is adequate. The reason of this behavior lies in the constraints on guards and invariants, which are not restrictive (i.e., they do not consist of “<" or “>", but consist of “<=", “>=" and “==").

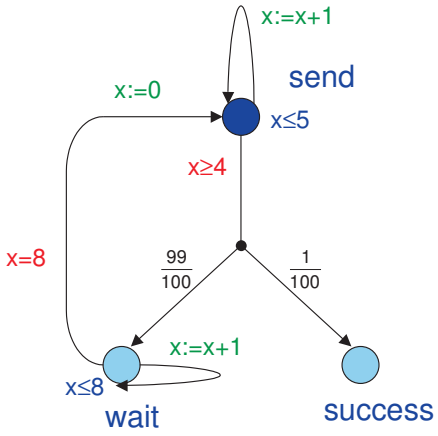


Figure 3.27: The integer semantic is sometimes adequate.

CHAPTER 4

Model Checking for e-commerce - Verification of Security Protocols

4.1 Introduction

Electronic commerce (e-commerce) became a buzzword as the information society developed rapidly throughout the 1990s. Internet has made e-commerce available to a wider user group, notably smaller enterprises and households. Amongst the business community the search for increased productivity and efficiency is expected to lead to even more enterprises adopting e-commerce as a way of doing business in the future. Whilst an ever growing awareness of the opportunities, technological developments in infrastructure and access devices, and falling access costs will facilitate this, fears about security and a lack of skills could hold it back.

According to EITO estimates, e-commerce on the Internet was valued at 172 billion EUR in 2001 in European Union, close to 2% of GDP (Gross Domestic Product). In that way safety errors on e-commerce could be very expensive.

But we can use *system validation* in order to avoid them [23, 125]. *System validation* is the process of determining the correctness of specifications, designs and products. Furthermore, it is a technique to support the quality control of the system design. A technique that implements *system validation* is **Model Checking** [105]. The system validation is an important goal in order to avoid design errors. Thus, in e-commerce must be a goal too. Verification allows us to check if our protocols hold the expected behaviors and if they hold some safety properties.

4.2 Timed Requirements of e-commerce Security Protocols

Any distributed system is a real time system where time manage must be considered as a strong constraint. For instance, in a distributed database, the time response for storing, updating or querying data determines whether the database is suitable or not.

E-commerce systems like any other distributed system consist of several parties and these parties interact among themselves in order to perform a purchase of products, services or others.

Thus, in e-commerce systems, time responses and the passage of time play also a crucial role. For instance, no body can accept a purchase process where the payment instant is abusive, or vice versa, any customer would accept a purchase process where the product reception instant is also abusive. - Imagine a lottery participant would make the payment once the lottery has finished, or even, a travel agent that sends the tickets once the fly has departed.

However, most of the methods used for the formal analysis of security protocols [38, 46, 55, 104] do not take time aspects into account. The main reason is that ignoring time simplifies the analysis, at the cost of making the analysis less realistic. Thus, in recent years, some related works, as Corin's et al. in [39], have appeared to deal with this problem.

Then it is necessary to take into account some requirements for designing and implementing security protocols to make them appropriate for time management. In this sense we must consider the following priority requirements:

1. The passage of time could affect the flow of messages. For instance, when an acknowledge message does not arrive in a suitable period of time, then it is necessary to retransmite again the message.

2. The protocol messages must include time information, because, Time information can be used in the protocol messages (e.g. timestamps).

The influence of time on the flow of messages is not considered by state of the art methods for analysing protocols. However, we believe it to be crucial because (1) If the protocol does not decide what action to take in the case of timeouts, the implementation will eventually have to consider these issues; (2) The efficiency and security of the implementation depends critically on these specific decisions; and (3) The timing of message flows in a protocol can be exploited by an attacker. Making judicious use of timing information in a protocol has received attention but mostly in the limited setting of using time stamps as opposed to nonces. Time information can be used to influence message flows as well. Summarising, we believe that timing issues are an important and hitherto insufficiently studied aspect of the design and implementation of security protocols.

4.3 Verification of e-commerce Security Protocols

E-commerce is based on transactions between client and server agents. These transactions require a protocol that provides privacy and reliability between these two agents. Two widely used protocols on e-commerce are Transport Layer Security (TLS) and Secure Electronic Transaction (SET). In this chapter we will see the way to use Model Checking to ensure the main e-commerce properties related to these protocols. Specifically, we use the tool UPPAAL to describe and analyze the behaviors of the protocols.

4.3.1 TLS protocol

The Transmission Control Protocol/Internet Protocol
(TCP/IP) governs the transport and routing of data over Internet. Other protocols, such as the HyperText Transport Protocol (HTTP), Lightweight Directory

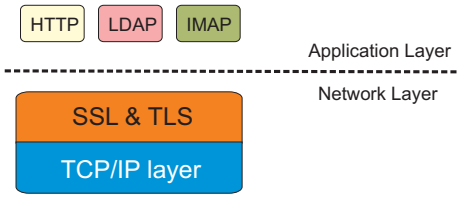


Figure 4.1: TLS runs above TCP/IP and below high-level application protocols

Access Protocol (LDAP), or Internet Messaging Access Protocol (IMAP), run "on top of" TCP/IP, in the sense that they all use TCP/IP to support typical application tasks such as displaying web pages or running email servers.

The Transport Layer Security protocol [58], TLS for short, runs above TCP/IP and below higher-level protocols such as HTTP or IMAP. It uses TCP/IP on behalf of the higher-level protocols, and it allows a TLS-enabled server to authenticate itself to an TLS-enabled client, it allows the client to authenticate itself to the server, and it allows both machines to establish an encrypted connection. These capabilities address fundamental concerns about communication over Internet and other TCP/IP networks:

TLS server authentication allows a user to confirm a server's identity. TLS enabled client software can use standard techniques of public-key cryptography to check that a server's certificate and public ID are valid and have been issued by a certificate authority (CA) listed in the client's list of trusted CAs. This confirmation might be important if the user, for instance, is sending a credit card number over the network and wants to check the receiving server's identity.

TLS client authentication allows a server to confirm a user's identity. Using the same techniques as those used for server authentication, TLS-enabled server software can check that a client's certificate and public ID are valid and have been issued by a certificate authority (CA) listed in the server's list of trusted CAs. This confirmation might be important if the server, for instance, is a bank sending confidential financial information to a customer and wants to check the recipient's identity.

An encrypted TLS connection requires all information sent between a client and a server to be encrypted by the sending software and decrypted by the

receiving software, thus providing a high degree of confidentiality. Confidentiality is important for both parties to any private transaction. In addition, all data sent over an encrypted TLS connection is protected with a mechanism for detecting tampering–that is, for automatically determining whether the data has been altered in transit.

The TLS Record Protocol is used for encapsulation of various higher level protocols. One such encapsulated protocol, the TLS Handshake Protocol, allows the server and client to authenticate each other and to negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data. The TLS Handshake Protocol provides connection security that has three basic properties:

- The peer's identity can be authenticated using asymmetric, or public key cryptography (e.g., RSA, DSS, etc.). This authentication can be made optional, but is generally required for at least one of the peers.

- The negotiation of a shared secret is secure: the negotiated secret is unavailable to eavesdroppers, and for any authenticated connection the secret cannot be obtained, even by an attacker who can place himself in the middle of the connection.

- The negotiation is reliable: no attacker can modify the negotiation communication without being detected by the parties of the communication.

The TLS Handshake Protocol

The cryptographic parameters of the session state are produced by the TLS Handshake Protocol, which operates on top of the TLS Record Layer. When a TLS client and server first start communicating, they agree on a protocol version, select cryptographic algorithms, optionally authenticate each other, and use public-key encryption techniques to generate shared secrets.

The TLS Handshake Protocol involves the following steps:

- Exchange hello messages to agree on algorithms, exchange random values, and check for session resumption.

- Exchange the necessary cryptographic parameters to allow the client and server to agree on a premaster secret.
- Exchange certificates and cryptographic information to allow the client and server to authenticate themselves.
- Generate a master secret from the premaster secret and exchanged random values.
- Provide security parameters to the record layer.
- Allow the client and server to verify that their peer has calculated the same security parameters and that the handshake occurred without tampering by an attacker.

The handshake protocol can be summarized as follows: The client sends a client hello message to which the server must respond with a server hello message, or else a fatal error will occur and the connection will fail. The client hello and server hello are used to establish security enhancement capabilities between client and server. Following the hello messages, the server will send its certificate if it is to be authenticated. If the server is authenticated, it may request a certificate from the client. Now the server will send the server hello done message, indicating that the hello-message phase of the handshake is complete. The server will then wait for a client response. If the server has sent a certificate request message, the client must send the certificate message. The client key exchange message is now sent, and the content of that message will depend on the public key algorithm selected between the client hello and the server hello. If the client has sent a certificate with signing ability, a digitally-signed certificate verify message is sent to explicitly verify the certificate. At this point, a change cipher spec message is sent by the client, and the client copies the pending Cipher Spec into the current Cipher Spec. The client then immediately sends the finished message under the new algorithms, keys, and secrets. In response, the server will send its own change cipher spec message, transfer the pending to the current Cipher Spec, and send its finished message under the new Cipher Spec. At this point, the handshake is complete and the client and server may begin to exchange application layer data. We can see the

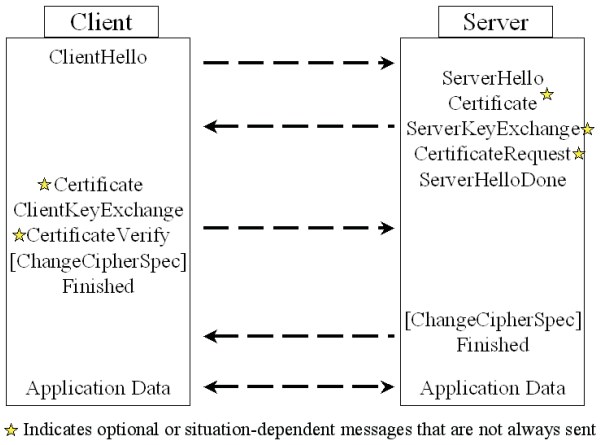


Figure 4.2: Message flow for a full handshake

corresponding flow chart in Figure 4.2. Note that the ChangeChipherSpec is an independent TLS Protocol content type, and it is not actually a TLS handshake message.

Modeling the Protocol

Now we have presented the main features of UPPAAL models. Then, let us see how we can describe the TLS Handshake protocol by means of timed automata. In this task we firstly identify two processes in the protocol: the Client and the Sever processes.

Once we have identified the protocol processes, we model the message flow between them and their internal behaviour. The message flow is described by means of synchronizations between the Client and the Server (figures 4.3 and 4.4, respectively). The internal behaviour of each process can be easily modeled, by some "local" state and transitions. Note that the "Anonymous" message is really a server "KeyExchange" message. But this "KeyExchange" message is sent in an anonymous negotiation. Furthermore, the "HandShakeFailure" message represents the possible errors.

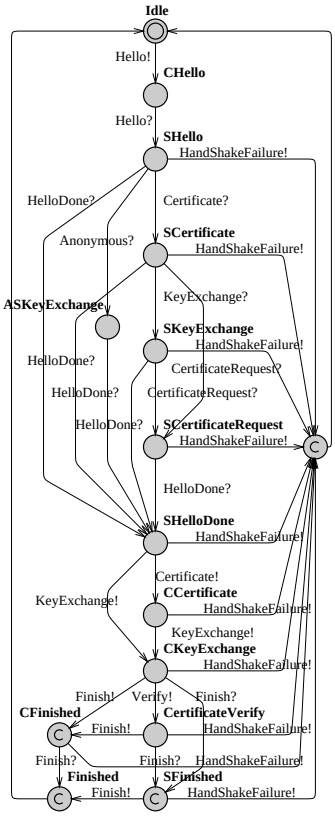


Figure 4.3: The Client process

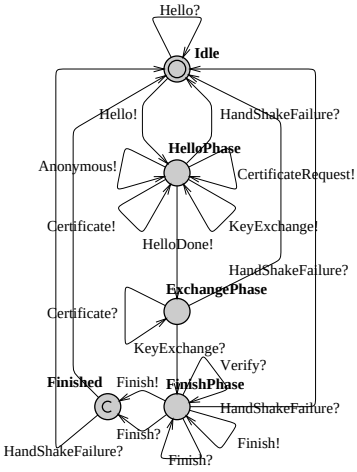


Figure 4.4: The Server process

Validating the Protocol

In the validating phase we can check whether the model holds the system behavior or not. This can partially be made by means of simulations. These are made by choosing different transitions and delays along the system evolution. At any moment during the simulation, you can see the variable values and the enabled transitions. Thus, you can choose the transition that you want to execute. Nevertheless, you can also select the random execution of transitions, and thus, the system evolves by executing transitions and delays which are selected randomly. We have some other options in the Simulator. For instance, you can save simulations traces that can later be used to recover an specific execution trace. Actually, the simulation is quite flexible at this point, and you can back or forward in the sequence.

Then, with respect to our model of the TLS Handshake protocol, our main goal in the validation phase is to check the correctness of the message flow, taking into account the protocol definition.

We have made a number of simulations; and we have concluded that the system design satisfies the expected behavior in terms of the message flow between the Client and the Server. For instance, we show, in Figure 4.5, the

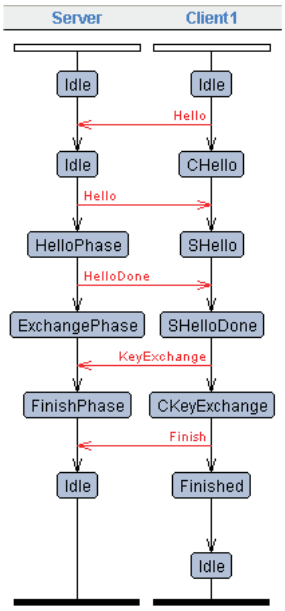


Figure 4.5: Trace for an abbreviated Handshake

negotiation trace for an abbreviated handshake.

Specifying and Verifying Properties

Before starting the automatic verification, we must establish which are the properties that the model must fulfill.

We have divided these properties into three classes: Safety, Liveness and Deadlocks. These properties are specified by means of a *Temporal Logic*. The temporal Logic used by UPPAAL is described in [92].

Safety Properties allow us to check if our model satisfies some security restrictions. The main Safety properties are:

- A Server could not send the *ServerHello* message if the client has not sent the *ClientHello* message before, or vice versa:

$$\forall \Box Client.SHello \Rightarrow Server.HelloPhase \tag{4.1}$$

- A Client sends the *KeyExchange* message immediately after the server *certificate* message (or *ServerHello* message, if this is an anonymous negotiation):

$$\forall\Diamond(Client.CKeyExchange \Rightarrow Server.HelloPhase) \quad \vee$$
$$(Client.CKeyExchange \Rightarrow Server.ExchangePhase) \tag{4.2}$$

- The *Finished* message is always sent by the sender and Client, once the key exchange and the authentication processes are successful:

$$\forall\Box Client.Finished \Rightarrow Server.FinishPhase \tag{4.3}$$

Liveness Properties intend is to check that our model can evolve in the right order. Liveness Properties for our model are simple. If a client sends the message *ClientHello*, some time later, we could reach the message *finished*. Translating it into Temporal Logic we have:

$$Client.CHello \longrightarrow Client.Finished \tag{4.4}$$

On the other hand, if a server sends a ServerHellow, some time later, we could reach the message finished. Translating it into Temporal Logic we have:

$$Server.HelloPhase \longrightarrow Server.Finished \tag{4.5}$$

Deadlocks are clear restrictions. We could check if our model is deadlocks free:

$$\forall\Box\neg Deadlock \tag{4.6}$$

Thus, we have specified the properties in UPPAAL and we have checked them by the UPPAAL verifier. The verifier outputs that we have obtained allow us to conclude that the safety properties 4.1, 4.2, 4.3 and the deadlock freeness property 4.6 are held by our model.

But we have obtained a negative answer in the liveness properties 4.4 and 4.5. By analyzing these properties, we supposed that when a *hello* message has

arrived, the message *finished* will be sent later. But sometimes, the client or the server can send the *Handshake-Failure* message, which means "something wrong has happened" and then the negotiation ends. Then, we can replace the properties 4.4 and 4.5 by: When the hello message has been sent, sometimes the negotiation ends. It is expressed by 4.7 and 4.8, which are held by our model.

$$\forall\Diamond Client1.CHelloimplyServer.FinishPhase \tag{4.7}$$

$$\forall\Diamond Server.HelloPhaseimplyClient1.Finished \tag{4.8}$$

4.3.2 The Set Protocol

Secure Electronic Transaction (SET) is a system for ensuring the security of financial transactions on the Internet. It was supported initially by Mastercard, Visa, Microsoft, Netscape, and others. With SET, a user is given an electronic wallet (digital certificate) and a transaction is conducted and verified using a combination of digital certificates and digital signatures among the purchaser, a merchant, and the purchaser's bank in a way that ensures privacy and confidentiality. SET makes use of Netscape's Secure Sockets Layer (SSL), Microsoft's Secure Transaction Technology (STT), and Terisa System's Secure Hypertext Transfer Protocol (S-HTTP). SET uses some but not all aspects of a public key infrastructure (PKI). SET works in the following way:

Let us assume that a customer has a SET-enabled browser, such as Netscape or Microsoft's Internet Explorer, and that the transaction provider (bank, store, etc.) has a SET-enabled server.

1. The customer opens a Mastercard or Visa bank account. Any issuer of a credit card is some kind of bank.

2. The customer receives a digital certificate. This electronic file functions as a credit card for online purchases or other transactions. It includes a public key with an expiration date. It has been through a digital switch to the bank to ensure its validity.

3. Third-party merchants also receive certificates from the bank. These certificates include the merchant's public key and the bank's public key.

4. The customer places an order over a Web page.

5. The browser of the customer receives and confirms that the merchant is valid from its certificate.

6. The browser sends the order information. This message is encrypted with the merchant's public key, the payment information, which is encrypted with the bank's public key (which cannot be read by the merchant), and information that ensures the payment can only be used with this particular order.

7. The merchant verifies the customer by checking the digital signature on its certificate. This may be done by referring the certificate to the bank or to a third-party verifier.

8. The merchant sends the order message to the bank. This includes the bank's public key, the customer's payment information (which the merchant cannot decode), and the merchant's certificate.

9. The bank verifies the merchant and the message. The bank uses the digital signature on the certificate with the message and verifies the payment part of the message.

10. The bank digitally signs and sends authorization to the merchant, who can then fill the order.

The set of participants in a SET protocol are represented in figure 4.6.

Cardholder A cardholder uses a payment card that has been issued by the Issuer. SET ensures that the interactions of cardholder with the merchant and the payment card information remain confidential.

Issuer It is a financial institution. It establishes an account for the cardholder. It also issues the payment card. The Issuer also guarantees payment for authorized transactions using the payment card.

Merchant The merchant offers goods or services in exchange for a payment. In order to accept payment cards the merchant must have a relationship with an Acquirer.

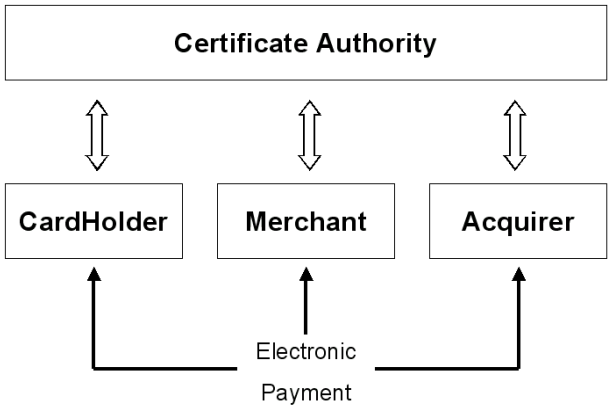


Figure 4.6: Participants in a SET protocol & their Interactions.

Acquirer A financial institution that establishes an account with a merchant and process payment card authorizations and payments.

Payment Gateway It is a device operated by an Acquirer or a designated third party that processes merchant payment messages, including payment instructions from cardholders.

Brand Financial Institutions came up with payment card brands in order to protect and advertise the brand. It creates an atmosphere conducive to establishing and enforcing rules for use and acceptance of their respective payment cards. It also provides a network to interconnect the financial institutions.

Third Parties Issuers and Acquirers sometime assign processing of payment card transactions to these third-party processors. [134]

Once we have described the protocol, we can validate the protocol with UPPAAL and RAPTURE, after translating it.

Modelling, Validating, Translating and Verifying in UPPAAL and RAPTURE

To **model** this protocol in UPPAAL we have used the formal description that is available at http://www.setco.org [134, 135, 136]. In figures 4.7,4.8,4.9 and 4.10 we can see the main processes involved in the protocol.

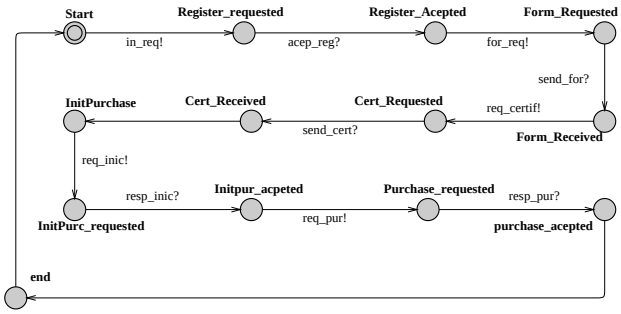


Figure 4.7: CardHolder Template.

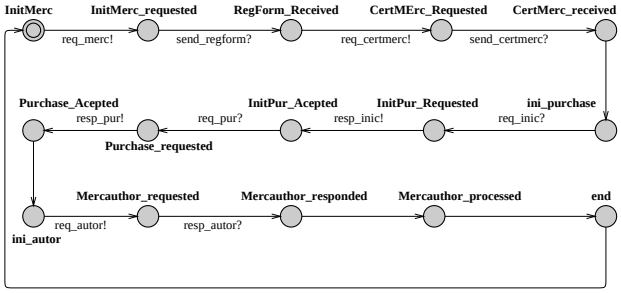


Figure 4.8: Merchant Template.

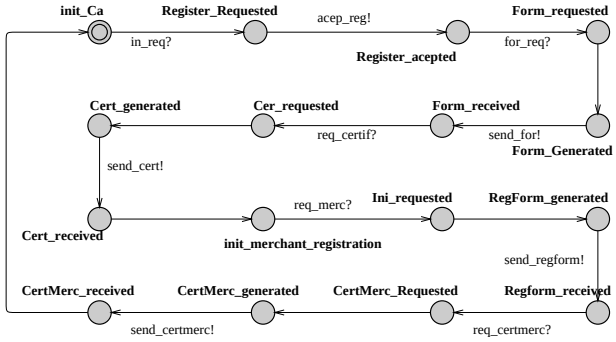


Figure 4.9: Certificate Authority Template.

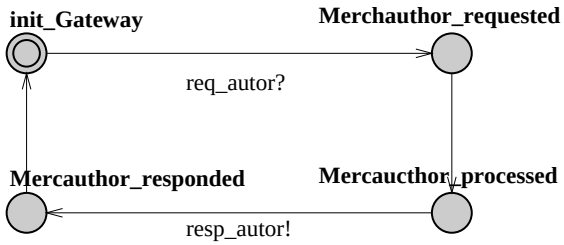


Figure 4.10: Payment Gateway Template.

Then, we start by checking that the model we have constructed satisfies the expected behavior. This task is accomplished by simulation. Simulations are made by choosing different transitions and delays along the system evolution. At any moment during the simulation, we can see the variable values and the enabled transitions. Thus, we can choose the next transition to be executed. But we have also the possibility to make an automatic execution, and thus, the system evolves by executing transitions and delays which are selected randomly. Then, with respect to our model of the SET protocol, our main goal in the validation phase is to check the correctness of the message flow, taking into account the protocol definition. We have made a number of simulations; and we have concluded that the system design satisfies the expected behavior in terms of the message flow between the Client and the Server.

To **verify** the protocol in UPPAAL we must establish the properties that the model must fulfill. We have divided these properties into three classes: Safety, Liveness and Deadlocks. These properties are specified by means of a *Temporal Logic*. The temporal Logic used by UPPAAL is described in [92].

Safety Properties allow us to check if our model satisfies some security restrictions. In our case study the main safety properties are:

- A Merchant cannot start the identification process if CardHolder has not started the identification process:

$$\forall\Box CardHolder.RegisterRequested \Rightarrow$$
$$Merchant.InitMercRequested \tag{4.9}$$

- A Certificate Authority sends the registration form immediately after the Merchant starts the identification:

$$\forall\Diamond(Merchant.InitMercRequested \Rightarrow$$
$$CertificateAuthority.RegFormReceive) \tag{4.10}$$

- A Payment Gateway cannot finish authorization if the Merchant has not requested:

$$\forall\Box Merchant.InitAuthority \Rightarrow$$
$$PaymentGateway.MerchAuthorResponded \tag{4.11}$$

Liveness Properties check that our model can evolve in the right order. Liveness Properties for our model are simple. If a CardHolder sends the message *Register Request*, some time later, we must obtain the message *Purchase Accepted*. Translating this into Temporal Logic we obtain:

$$CardHolder.RegisterRequested \longrightarrow$$
$$CardHolder.PurchaseAccepted \tag{4.12}$$

On the other hand, if a Merchant sends a registered request, some time later, the Certificate Authority sends a certificate.

$$CertificateAuthority.RegisterRequested \longrightarrow$$
$$CertificateAuthority.CertMercReceived \tag{4.13}$$

Deadlocks are clear restrictions, which can be checked by:

$$\forall\Box\neg Deadlock \tag{4.14}$$

These properties has been verified by the UPPAAL verifier engine with an affirmative response to them.

CHAPTER 5

Design and Verification of Web Services

5.1 Introduction

A Web Service is an autonomous, standards-based component whose public interfaces are defined and described using XML [85]. Other systems may interact with a Web Service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols. Web Services specifications offer a communication bridge between the heterogeneous computational environments used to develop and host applications. The future of E-Business applications requires the ability to perform long-lived, peer-to-peer collaborations between the participating services, within or across the trusted domains of an organization.

The Web Service architecture stack targeted for integrating interacting applications consists of the components shown in Fig. 5.1.

The Web Services Choreography specification [85] is aimed at the composition of interoperable collaborations between any type of party regardless of the supporting platform or programming model used by the implementation of the hosting environment. Figure 5.2 illustrates the relationship between WS-CDL, the choreography layer and the orchestration layer (WS-BPEL), taking

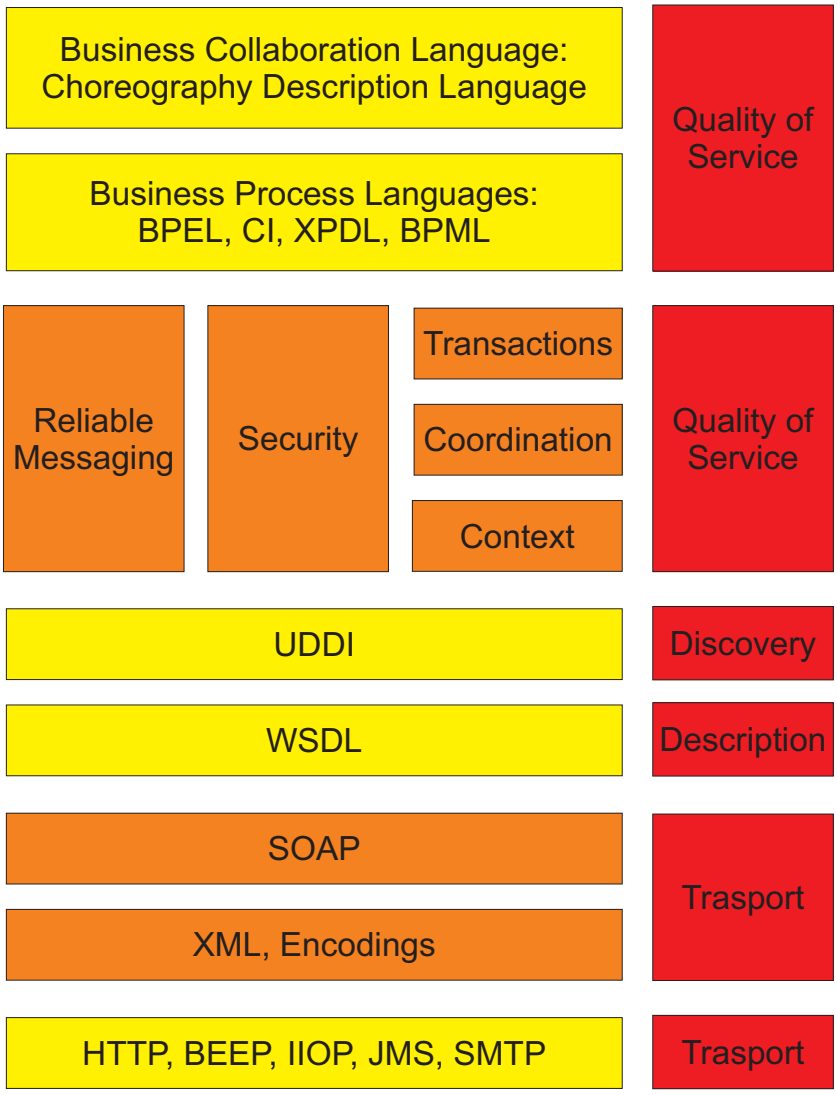


Figure 5.1: The Stack Protocol for Web Services

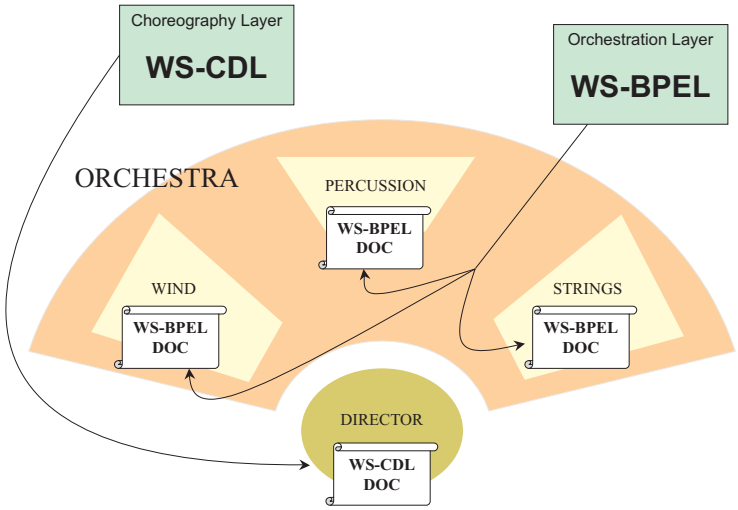


Figure 5.2: Relationship between WS-CDL and WS-BPEL.

an orchestra as a metaphor of this relation. The key document is the director score, which corresponds to the WS-CDL document, in which each participant is represented as well as the time it enters into action. Furthermore, the wind, percussion and strings scores correspond to the WS-BPEL documents, which show the behavior of each particular group.

In Web Services the design phase is the most important phase for Choreography and Orchestration layers. These layers describe the behaviors of each participant in a Web Service. In brief the Choreography describes the general behavior and the Orchestration describes each particular behavior. Thus an important goal is the validation and Verification of these layers in order to proof their correctness.

The verification process must play an important role in Web Services development. However, it is necessary to notice that due to the wide range of systems that Web Services are applied to, it is mandatory to check the different systems characteristics and a critical characteristic of numerous systems is related with time constraints. Thus, it becomes important for Web Services frameworks to ensure the correctness of systems with time constraints. For instance, we can

think in a failure of a bank to receive a large electronic funds transfer on time, which may result in huge financial losses. Then, there is growing consensus that the use of formal methods, development methods based on some formalism, could have significant benefits in developing E-business systems due to the enhanced rigor these methods bring [70]. Furthermore, these formalisms allow us to reason with the constructed models, analysing and verifying some properties of interest of the described systems.

5.2 Designing Correct Web Services

5.2.1 What is a “correct´´ Web Service?

Before introducing the reader in our methodology proposal, we must define what is a “correct Web Service":

> We say that a Web Service is correct if it is well designed, in the sense that it has been checked for potential design errors.

We classify the potential design errors into different types, specific and general, which are defined as follows:

1. Specific, i.e., errors that are checked via properties that are *specific to the current application.*

2. General, i.e., errors related to general characteristics such as concurrency, quality of service, time response, security, etc.

5.2.2 How can we generate “correct´´ Web Services?

In the generation of Web Services [50] as in the generation of any software system it is necessary to apply a methodology covering every phase of the life cycle. The methodology that we propose consists of the following phases:

- **System analysis:** In this first phase the goal is to capture and specify the requirements that the system must fulfil.

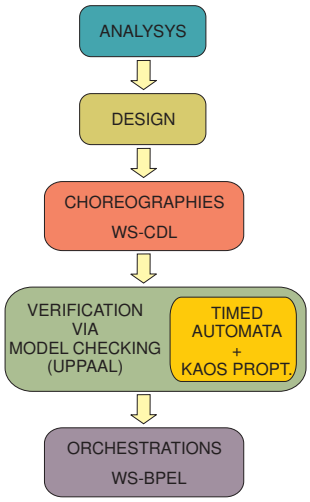


Figure 5.3: Proposed methodology for the design of “correct" WS.

- **System design:** In this phase we use the provided information from the previous phase in order to design the system.

- **Model generation & verification:** This phase is focused on the modeling and verification of the designed system in order to prove its correctness. The properties to verify are captured in the *System analysis* phase.

- **Web Services generation:** Once the design have been checked, Web Services are automatically generated, by using some tool.

In the analysis phase we have used a technology based on goal models performing *requirement engineering*, which is named KAOS. In the design phase we use the Unified Modeling Language (UML) 2.0. These UML diagrams are used to describe and specify the choreographies, which will be translated into timed automata for verification purposes. Figure 5.3 depicts a diagram that shows the different phases of the proposed methodology.

5.3 System Analysis

The requirements of the system are collected in this phase. However, they must be expressed in a standardized manner. There are several languages, graphical diagrams, etc. to perform it, but we apply those that allow us to capture in a proper way the time requirements. Thus, in this phase we use the goal-oriented requirements engineering.

The key activity in goal-oriented requirements engineering is the construction of the goal model. Goals are objectives the system under construction must achieve. Goal formulations thus refer to intended properties to be ensured. They are formulated at different levels of abstraction from high-level, strategic concerns to low-level technical concern.

Goal models also allow analysts to capture and explore alternative refinements for a given goal. The resulting structure of the goal model is an AND-OR graph. The specific goal-oriented framework considered here is the KAOS methodology [41, 117, 44, 89] which uses a two level language: (1) an outer semi-formal layer for capturing requirements engineering concepts, structuring and presenting them; (2) an inner formal assertion layer for their precise definition and for reasoning about them.

A goal is a concrete requisite about some system that requires the cooperation between different parties, which are called *agents*. Agents are active components that play a role towards goal satisfaction. Goals may refer to services to be provided (functional goals) or to the quality of service (non-funcional goals).

To Build Goal Models, Goals are organized in AND/OR refinement-abstraction hierarchies where higher-level goals are in general strategic, coarse-grained and involve multiple agents whereas lower-level goals are in general technical, fine-grained and involve fewer agents. In such structures, AND-refinement links relate a goal to a set of subgoals (called refinement) possibly conjoined with domain properties; this means that satisfying all subgoals in the refinement is a sufficient condition in the domain for satisfying the goal, (Figure 5.4). OR-refinement links may relate a goal to a set of alternative refinements, (Figure 5.5).

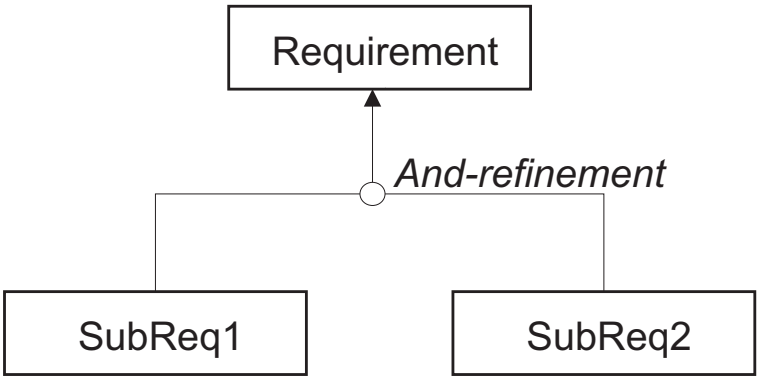


Figure 5.4: And-refinement goal model

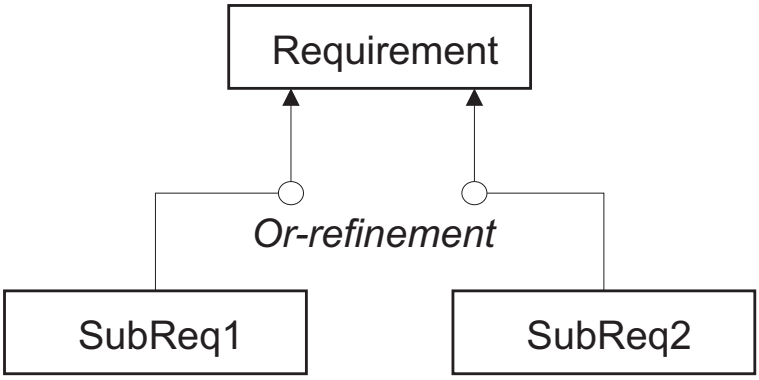


Figure 5.5: Or-refinement goal model

A requirement is a terminal goal under responsibility of an agent in the software-to-be; an expectation is a terminal goal under responsibility of an agent in the environment. Goals prescribe intended behaviors; they are optionally formalized in a real-time temporal logic that we have shown before in subsection 3.2.3. Keywords such as Achieve (reachability), Avoid (not safety), Maintain (safety), possibly always, inevitably and unbounded response, are used to name goals according to the temporal behavior pattern they prescribe. They are depicted in the goal model as follows:

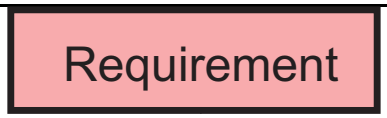
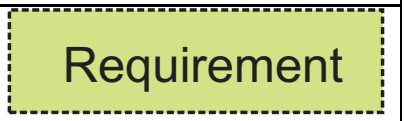
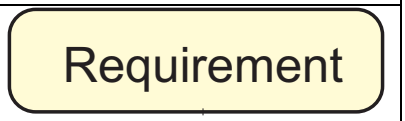
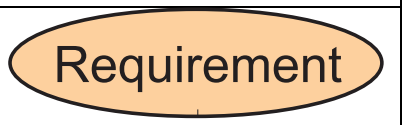
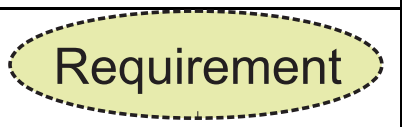
Temporal Behavior	**Goal Model Representation**
Maintain (Safety) $A[\ ]\ \varphi$	
Achieve (Reachability) $E <>\ \varphi$	
Possibly Always $E[\ ]\ \varphi$	
Inevitably $A <>\ \varphi$	
Unbounded Response $\varphi -- >\ \psi$	

Figure 5.6 shows a goal model fragment of an aircraft control system. The goal Maintain (safety) [SafeDoors] is refined by two *OR-refinement* links that inherit the safety behavior. The first, leaf goal Maintain[DoorsLockedWhileMoving], may be annotated with the following temporal logic assertion stating that in every future state the plane doors shall be locked when the plane is moving: $A[\ ]\ (Plane.Moving \wedge Doors.Locked)$. The second, leaf goal Maintain Inevitably [DoorUnlockedWhenEmergency], may be annotated with te following temporal

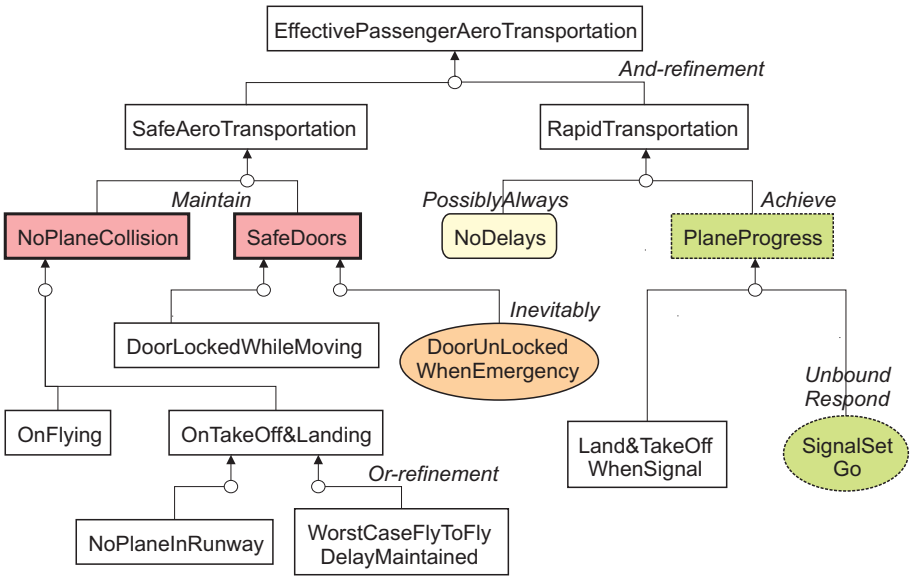


Figure 5.6: Portion of a goal graph for a aircraft control system

logic assertion stating that in every future state the plane doors shall be unlocked when an emergency occur: $A <> Plane.Emegency \wedge Doors.Unlocked$.

The alternative refinement for the safety subgoal [NoPlaneCollisionOnTake-Off&Landing] which leads to totally different designs when refined: block-based design for NoPlainIn - SameRunway and speed control design for WorstCase-FlyToFlyDelayMaintained,
$A[]\ \neg(Runway.Ocduppied)$ and
$A[]\ Runway.FlyToFlyClock < FlyToFlyDelay$, respectively.

The leaf goal [NoDelays] specifies the intended behavior of "Possibly Always, the aircraft control does not suffer delays" that is defined by the formula
$E[]\ \neg(AircraftControlSystem.Delay)$

The temporal behavior for the progression of planes is specified by the Achieve (reachability) subgoal [PlaneProgress] that is refined by two subgoals that inherit the achieve behavior: [Land&TakeOff WhenSignal] and [SignalSetGo] defined by $E <> (Plane.LandorPlane.TakeOff) \wedge Signal.Go$ and
$E <> Plane.Ready -- > Signal.Go$

Goals refer to objects or entities which may be incrementally designed from goal specifications to produce a structural model of the system that we will see in the next subsection by using UML diagrams. Objects have states defined by the values of their attributes and associations to other objects. In the above formalization of the goal DoorsLockedWhileMoving, Moving and doorsState are attributes of the Train and doors entities that will be declared in the system design.

5.4 System Design

The system design phase is aimed in the system description that identify and capture each particular behavior. In software engineering the Unified Modeling Lenguaje (UML) diagrams is a well known tool for the design phase. UML 2.x [3] considers thirteen different diagrams for software development. They can be divided into three groups:

- *Behavior diagrams.* A type of diagram that depicts behavioral features of a system or business process. This includes activity, state machine, and use case diagrams as well as the four interaction diagrams.
- *Interaction diagrams.* A subset of behavior diagrams which emphasize object interactions. This includes communication, interaction overview, sequence, and timing diagrams.
- *Structure diagrams.* A type of diagram that depicts the elements of a specification that are irrespective of time. This includes class, composite structure, component, deployment, object, and package diagrams.

Our interest is focused on the behavior and interaction diagrams, because with them we are able to capture the main features of systems with time restrictions and, moreover, are easily translated to choreography and orchestration layers. More specifically, we are interested in the sequence and activity diagrams, because they can represent the message flow in time and the behavior of several processes, respectively.

These diagrams are defined as follows:

1. *"The **Sequence Diagram** models the sequential logic, in effect the time ordering of messages between classifiers."*

   Sequence Diagrams of UML 2.x (Object Management Group 2003) model the logic flow of the system in a visual manner, enabling the designer both to document and validate the logic. Sequence diagrams are used to capture the dynamic behavior of the system, together with other diagrams, like activity diagrams, communication diagrams, timing diagrams, and interaction overview diagrams.

   Figure 5.7 depicts an example of a sequence diagram. There are three participants in this diagram: alpha, beta and gamma. These entities are represented by using objects and the communication processes are represented by using different types of arrows request, response and request&response. The broken line that starts from each object represents the line of time. The labeled boxes choice1, choice2, parallel and workunit represent three different types for control. Choices one and two represent two options, only one of them could be performed. The parallel box represents interactions that could be performed in parallel. And finally, a workunit represents a set of interactions that are within a loop that has a guard and a repeat condition. In the second choice there is an edge labeled with *"Clock $x < 5$"*, representing an interaction that must occur before the clock x reaching the value of five units.

2. *"The **Activity Diagram** depicts high-level business processes, including data flow, or to model the logic of complex logic within a system."*

   Activity diagrams of UML 2.x (Object Management Group 2003) are typically used for business process modeling, for modeling the logic captured by a single use case or usage scenario, or for modeling the detailed logic of a business rule. In many ways UML activity diagrams are the object-oriented equivalent of flow charts and data flow diagrams (DFDs) from structured development.

   Figure 5.8 depicts an example of an activity diagram. The filled circle represents the beginning of one activity. The activities are represented by

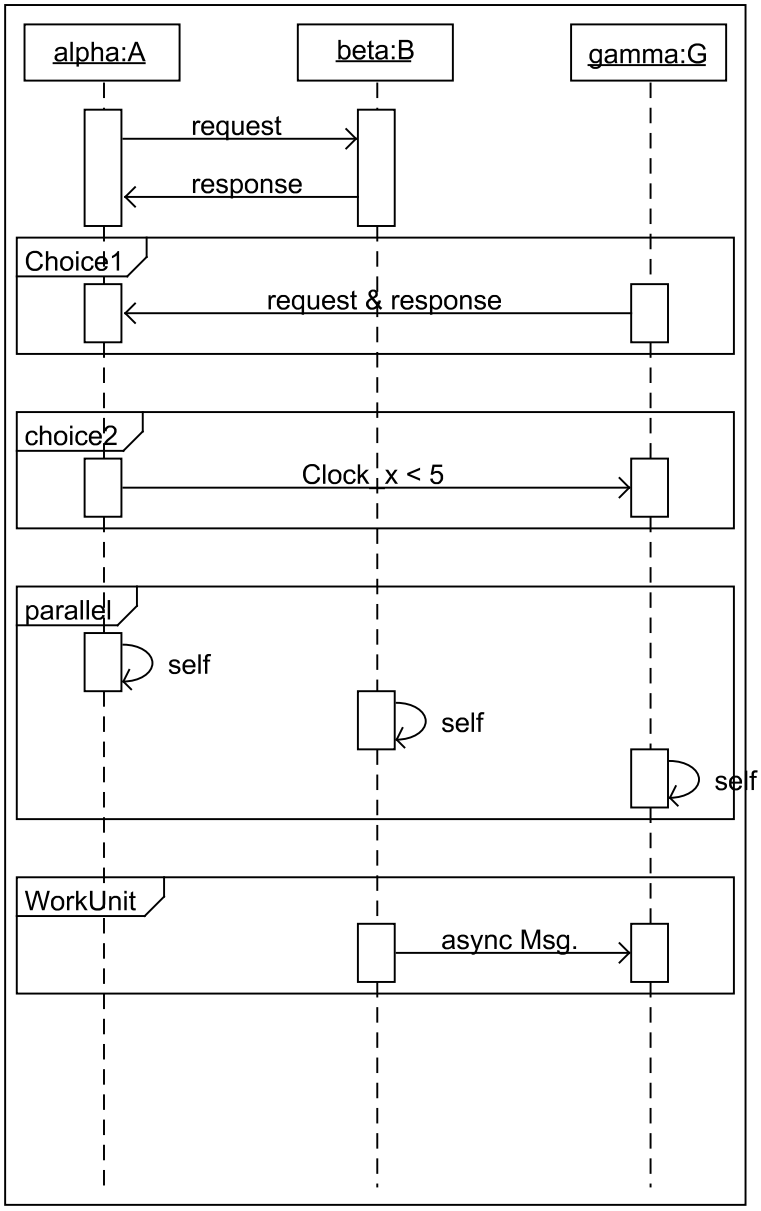


Figure 5.7: A sequence diagram with three objects

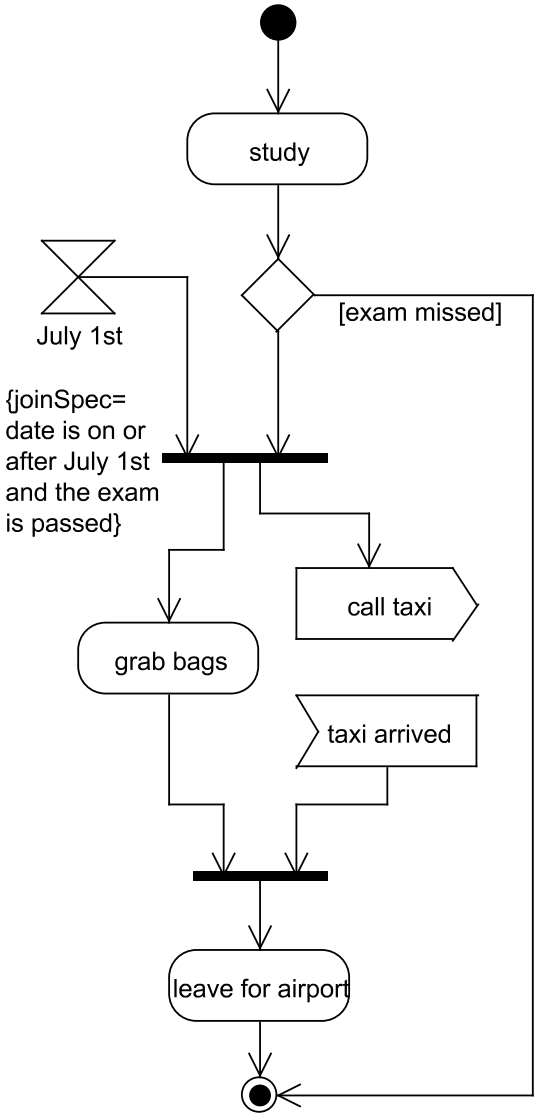


Figure 5.8: A basic activity diagram

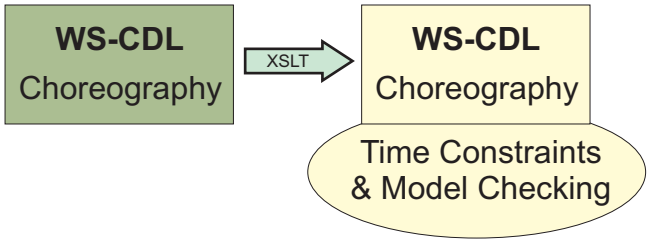


Figure 5.9: Translation from WSCDL into Timed Automata

using round boxes. Transitions between activities are depicted with arrows, and synchronizations by gross lines. A diamond shows two possible outputs from the activity *study*, either a positive or a negative output. The time restriction is captured by a clock that is active when the date is first of July. The boxes call taxi and taxi arrived are signals that are sent to environment entities. Concentric circles are used to represent the end of activities.

5.5 Translation from WS-CDL into Timed Automata

Figure 5.9 depicts the translation process. For this purpose, we must first analyze the WS-CDL documents in order to identify the common points shared between them. The first stage is to obtain the general structure describing the system that we are analyzing. In timed automata, this structure is defined by the so-called *System*, which consists of the individual processes that must be executed in parallel. Each one of these processes is defined by using a template. Templates are used to describe the different behaviors that are available in the system.

Then, for each component of a WS-CDL description we have the following correspondence in timed automata (see Fig. 5.10 for a schematic presentation of this correspondence):

Roles : These are used to describe the behavior of each type of party that we are using in the choreography. Thus, this definition matches with definition of a *template* in timed automata terminology.

Relation types : These are used to define the communications between two roles, and the channels needed for these communications. In timed automata we just need to assign a new channel for each one of these channels, which are the parameters of the templates that take part in the communication.

Participant types : These define the different parties that participate in the choreography. In timed automata they are processes participating in the system.

Channel types : A channel is a point of collaboration between parties, together with the specification of how the information is exchanged. As mentioned above, channels of WS-CDL correspond with channels of timed automata.

Variables : They are easily translated, as timed automata in UPPAAL support variables, which are used to represent certain information.

Now the problem is to define the behavior of each template. This behavior is defined by using the information provided by the flow of choreographies. Choreographies are sets of workunits or sets of activities. Thus, activities and workunits are the basic components of the choreographies, and they capture the behavior of each component.

Activities can be obtained as result of a composition of other activities, by using sequential composition, parallelism and choice. In terms of timed automata these operators can be easily translated:

- The sequential composition of activities is translated by concatenating the corresponding timed automata.
- Parallel activities are translated by the cartesian product of the corresponding timed automata.
- Choices are translated by adding a node into the automata which is connected with the initial nodes of the alternatives.

Finally, time restrictions are associated in WS-CDL with workunits and interaction activities. These time restrictions are introduced in timed automata

Role = Template
Relation Type = Channel$^+$
Participant Type = Process$^+$
Channel Type = Channel
Variables = Variables
Choreography = Choreography$^+$ | Activity
Activity = Work Unit | Sequence | Parallelism | Choice
Sequence = Activity$^+$
Parallelism = Activity$^+$
Choice = Activity$^+$
Work Unit = State & Guard & Invariant

where the symbols +, | are BNF notation,
em and & *is used to join information*

Figure 5.10: Schematic view of the translation

by means of guards and invariants. Therefore, in the case of a workunit of an activity having a time restriction we associate a guard to the edge that corresponds to the initial point of this workunit in the corresponding timed automaton.

In the appendixes we show the XSLT rules used for the translation process, as well as an WS-CDL example.

5.6 Case Studies

We introduce in this section two examples that clarify the methodology that we have described in the chapter. The first one is a purchase via Internet and the second one is a more complex system of a travel reservation system.

Some examples of the use of WS-CDL and WS-BPEL can be found in [22, 21, 51]. The second case study that we are going to use to illustrate how the translation works is closely linked to [51], where this particular case study was used to illustrate how timed automata can be used for the formal verification of properties for WS-CDL documents.

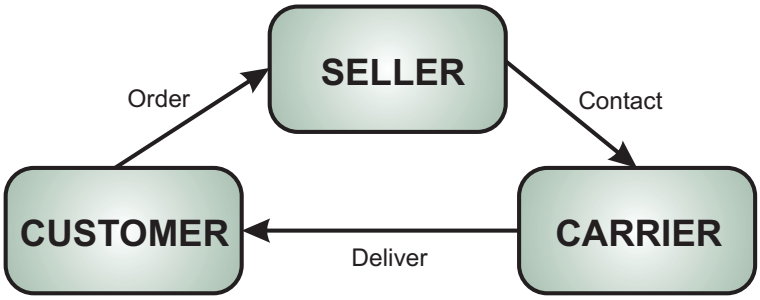


Figure 5.11: Internet Purchase Process

5.6.1 Basic Use Case: Internet Purchase

Use Case Description

This example is based upon a typical purchase process that uses Internet as business context for a transaction. There are three actors in this example: a customer, a seller and a carrier.

The Internet purchase works as follows: *"A customer wants to buy a product by using Internet. There are several sellers that offers different products in Internet Servers based on Web-pages. The customer contacts a seller in order to buy the desired product. The seller checks the stock and contacts with a carrier. Finally, the carrier delivers the product to the customer."*

Figure 5.11 depicts the diagram that represents this purchase process. This process consists of three participants: the customer, the seller and the carrier. The behavior of each participant is defined as follows:

- Customer: He contacts with the seller to buy a product. He must send to the seller the information about the product and the payment way. After the payment, he waits for receiving the product from a carrier within the agreed time, twenty four hours.

- Seller: He receives the customer order and the payment way. The seller checks if the stock is enought to deliver the order and sends an acceptance notification to the customer . If there is stock to deliver the order, then he contacts with a carrier to deliver the product.

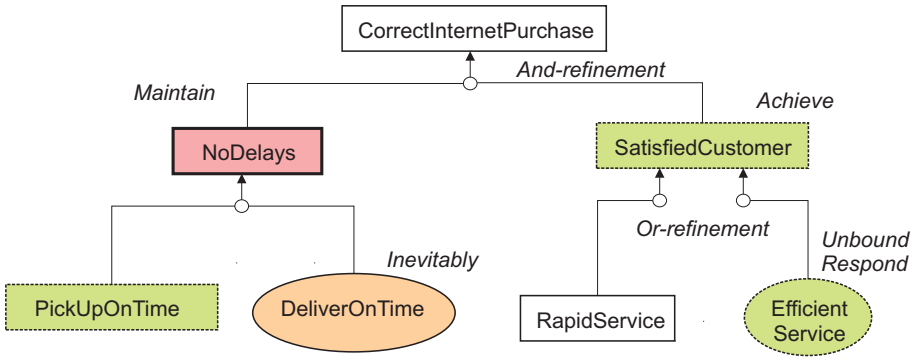


Figure 5.12: The goal-model for the Internet Purchase Process

- Carrier: He pickups the order and the customer information in order to deliver the product to the customer. The interval to deliver the product is the time that the seller has stipulated, one day.

Note that the main temporal constraint is the time to deliver the product. This interval starts when the seller accepts the order and finishes when the carrier delivers the product. The interval has been fixed on 24 hours, one day.

System Analysis

We must identify the crucial requirements for the Internet purchase process that we have described above. We have identified two different kinds of requirements. One kind refers to the obligation that both the seller and carrier have agreed to deliver the product on time, and the other one refers to the quality of service.

The time restriction establishes that the seller and carrier have twenty four hours to deliver the product. So, the seller must prepare the order for the carrier to send the product within the interval.

The service quality is determined by two different requirements that are close linked. The service must be rapid and also efficient. Due this close relationship between these two requirements if one of them is fulfilled then the other is fulfilled too.

Figure 5.12 depicts the goal-model that we have developed for this example. The root goal *"CorrectInternetPurchase"* is decomposed into two subgoals by

an And-refinement, which means that each one must be fulfilled in order to achieve the root goal.

The first one, *"NoDelays"*, that is of type "maintain", is refined by another And-refinement with two leaf goals that inherit the maintain character. The first leaf goal *"PickupOnTime"* is of type "Unbound Respond". This goal represents the situation that the carrier must pick up the order on time and is formalized as follows:

$$Customer.WaitOrder --> (Carrier.PickUp \wedge Clock_{deliver} < 24hours) \quad (5.1)$$

The second leaf goal *"DeliverOnTime"* is of type "Inevitably" and specifies that the carrier must deliver the order on time. The goal is defined as follows:

$$A <> (Carrier.Deliver \wedge Clock_{deliver} < 24hours) \quad (5.2)$$

The second one, *"SatisfiedCustomer"*, of type "Achieve", is formed by two leaf goals. These leaf goals refine the parent goal by an Or-refinement, which means that if one of them is satisfied then the parent goal is satisfied too. The leaf goal *"RapidService"*, that determines that the customer will receive the order on time, is specified as follows:

$$E <> (Customer.ReceiveOrder \wedge Clock_{deliver} < 24hours) \quad (5.3)$$

The leaf goal *"EfficientService"* has the behavior of an "Unbounded Response" requirement. This goal represents that when the seller accepts the order, then in the future, the customer will receive the order. This goal is formalized as follows:

$$Seller.AcceptOrder --> Customer.ReceiveOrder \quad (5.4)$$

System Design

In this phase we use the sequence and activity diagrams to represent the relationship between the participants and the particular behavior of each one of

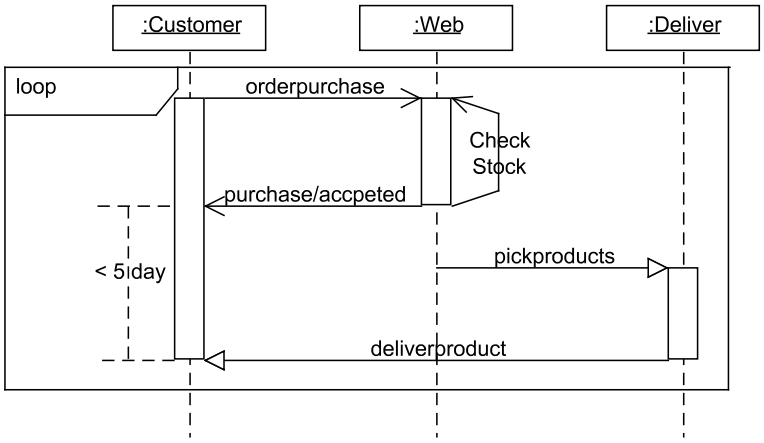


Figure 5.13: The sequence diagram for a purchase process by Internet

them. These diagrams are based on the requirements and system definition that we have seen before.

Figure 5.13 depicts the sequence diagram of our example. This diagram shows the messages exchanged between the three participants. Each message is labeled with the information that is exchanged during the communication process. The three participants are represented by using objects and the communication processes are represented by using different types of arrows.

The broken line that starts from each object represents the line of time. The interaction frame that is labeled with the word *loop* represents an entire locution between all the participants that is performed in an infinite loop. Finally, it is also possible to appreciate the time constraints between two messages, which is represented by using a broken line and a label describing the constraint.

Figure 5.14 depicts the activity diagram. This diagram shows the activities that are performed by the participants during the Internet purchase process. We can see three process separated by vertical lines, the costumer, the seller and the carrier.

The filled circle represents the beginning of one activity. The activities that each process perform are represented by using rounded boxes. Transitions between activities are depicted with arrows, and synchronizations by gross lines.

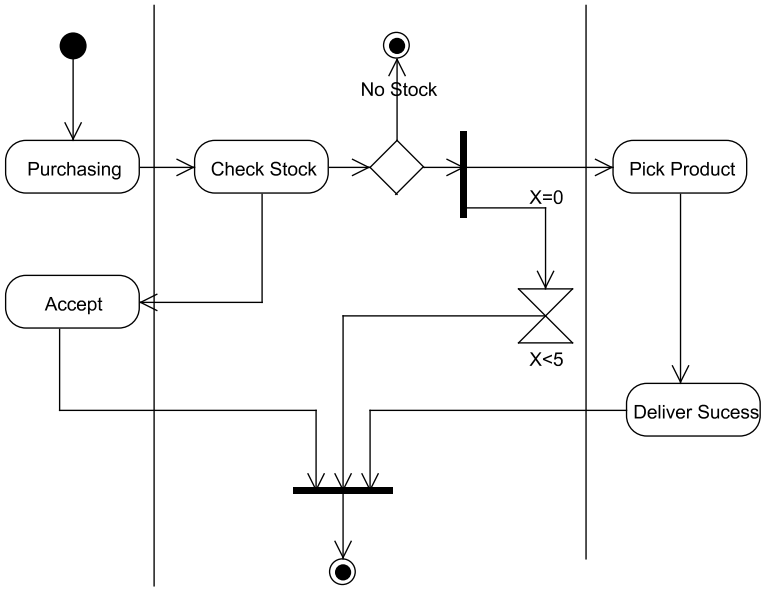


Figure 5.14: The activity diagram for a purchase process by Internet

A diamond shows two possible outputs from the activity *check stock*, either a positive or a negative output. The time restriction is captured by a clock that is reset when the *check stock* activity has finished with a positive response, and determines the interval in which the synchronization is available. Concentric circles are used to represent the end of activities.

WS-CDL Documents

The UML sequence and activity diagrams of Fig. 5.13 and 5.14 show the interaction and behavior of each one of the participants. The sequence diagram establishes the flow control of the purchase process (interactions, choices and parallelism), while the activity diagram establishes the concrete behaviors (time constraints, activities, etc).

Figure 5.15 shows a piece of the WS-CDL specification corresponding to this purchase process, which has been obtained from these UML diagrams. The complete specification is available in Appendix B.

```
<interaction name="createPO" channelVariable="tns:seller-channel"
             operation="handlePurchaseOrder" align="true" initiate="true">
  <participate relationshipType="tns:CostIntSellCarrRS"
               fromRole="tns:Customer" toRole="tns:Seller"/>
  <exchange name="request" informationType="tns:purchaseOrderType"
            action="request">
    <send variable="cdl:getVariable("tns:purchaseOrder", "", "")" />
    <receive variable="cdl:getVariable("tns:purchaseOrder", "", "")"
             recordReference="record-the-channel-info" />
  </exchange>
  <exchange name="response" informationType="purchaseOrderAccepted"
            action="respond">
    <send variable="cdl:getVariable("tns:purchaseOrderAcceted","","")"/>
    <receive variable="cdl:getVariable("tns:purchaseOrderAccepted","","")"/>
  </exchange>
  <exchange name="NoStockAckException" informationType="NoStockAckType"
            action="respond">
    <send variable="cdl:getVariable('tns:NoStockAck', '', '')"
          causeException="true" />
    <receive variable="cdl:getVariable("tns:NoStockAck","","")"
             causeException="true"/>
  </exchange>
  <record name="record-the-channel-info" when="after">
    <source variable="cdl:
                  getVariable("tns:purchaseOrder","","PO/CustomerRef")"/>
    <target variable="cdl:getVariable("tns:customer-channel", "", "")"/>
  </record>
  <record name="reset-clock" when="after">
    <source variable="00:00"/>
    <target variable="cdl:getVariable("tns:Clock1", "", "")"/>
  </record>
</interaction>
<interaction name="PickUpProductPO" channelVariable="tns:deliver-channel"
             operation="PickUpPurchaseOrder" align="true" initiate="true">
  <participate relationshipType="tns:CustIntSellCarrRS"
               fromRole="tns:Seller" toRole="tns:Carrier"/>
  <exchange name="request"
            informationType="tns:purchaseOrderType" action="request">
    <send variable="cdl:getVariable("tns:purchaseOrder", "", "")" />
    <receive variable="cdl:getVariable("tns:purchaseOrder", "", "")"
             recordReference="record-the-channel-info" />
  </exchange>
</interaction>
<interaction name="DeliverProductPO" channelVariable="tns:customer-channel"
             operation="DeliverProductOrder" align="true" initiate="true">
  <participate relationshipType="tns:CostIntSellCarrRS"
               fromRole="tns:Carrier" toRole="tns:Customer"/>
  <exchange name="request" informationType="tns:purchaseOrderType"
            action="request">
    <send variable="cdl:getVariable("tns:purchaseOrder", "", "")" />
    <receive variable="cdl:getVariable("tns:purchaseOrder", "", "")"
             recordReference="record-the-channel-info" />
  </exchange>
  <timeout  time-to-complete=
             "cdl:minor(cdl:getVariable("tns:Clock1","",""),"24:00")"/>?
</interaction>
```

Figure 5.15: WS-CDL interaction specification of the Internet purchase process

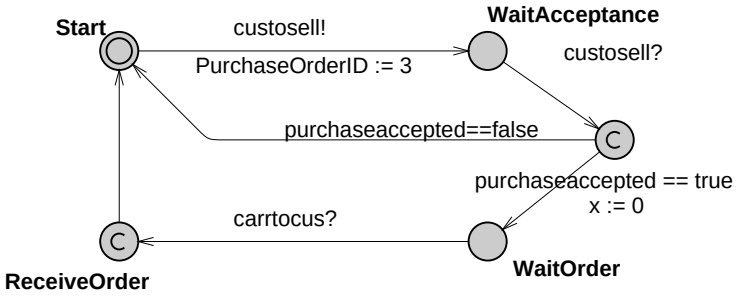


Figure 5.16: The customer automaton

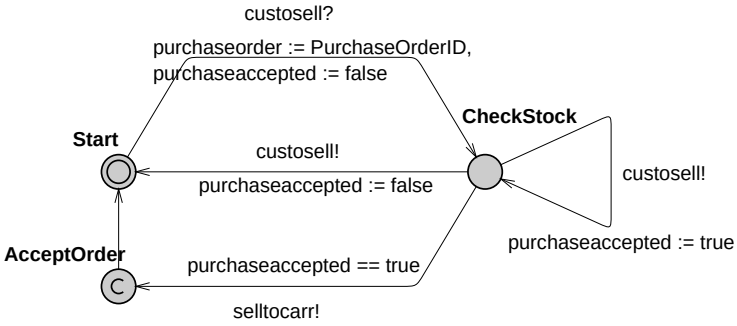


Figure 5.17: The seller automaton

Timed Automata Generation and Verification

In section 5 we introduce some rules for translating WS-CDL documents into timed automata (XML files that are accepted by UPPAAL). Thus, by applying these rules we obtain three timed automata: one corresponding to the **customer** (Fig. 5.16), another one to the **seller** (Fig. 5.17) and the last one to the **carrier** (Fig. 5.18).

Once we have obtained these timed automata, we can use the verifier of UPPAAL in order to check the properties that were identified. Notice that these properties must be adapted to consider the particular names of variables and clocks that are used in UPPAAL. For instance, the first property *"PickupOnTime"* (5.1) is rewritten as follows:

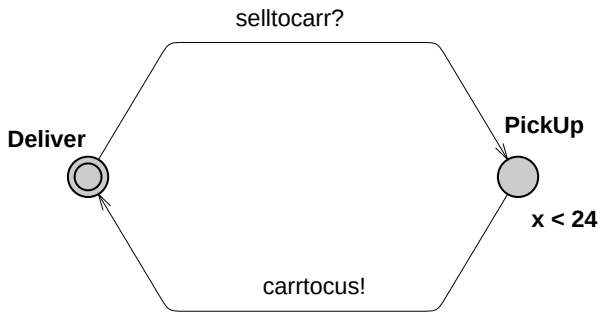


Figure 5.18: The carrier automaton

$$Customer.WaitOrder -- > (Carrier.PickUp \wedge x < 24) \tag{5.5}$$

The second property, *"DeliverOnTime"* (5.2) is rewritten as:

$$A <> (Carrier.Deliverandx < 24) \tag{5.6}$$

The third property *"SatisfiedCustomer"* (5.3) is rewritten as follows:

$$E <> (Customer.ReceiveOrderandx < 24) \tag{5.7}$$

The fourth property *"EfficientService"* (5.4) is rewritten as follows:

$$Seller.AcceptOrder -- > Customer.ReceiveOrder \tag{5.8}$$

Observe that the clocks $Clock_{deliver}$ is renamed to x.

We found an error in the verification of a property, concretely on Property 5.5 (Fig. 5.19). The problem appears when the seller sends the "acceptorder", but he does not send the "PickUp" message to the carrier within 24 hours. Then the carrier cannot deliver the product on time and the property is not fulfilled.

In order to correct this problem it is necessary to force the seller to send the "PickUp" message on time. For that purpose, we add an invariant to the seller state "CheckStock" labeled with $x < 2$. With this invariant the seller must send the message within 2 hours since he sent the message "PurchaseAccepted".

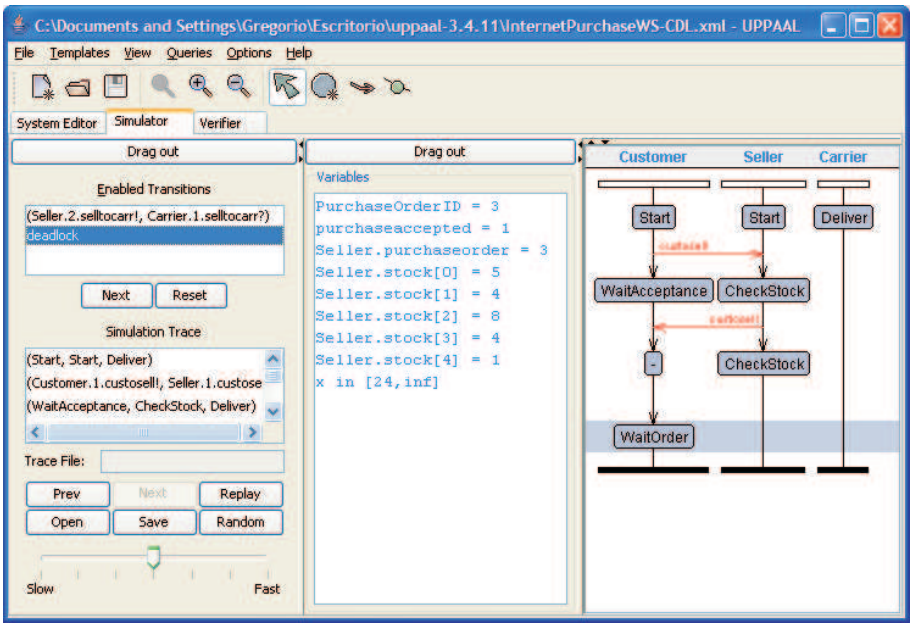


Figure 5.19: The Uppaal trace for property 5.5

Thus, the seller automaton would be replaced with the automaton depicted in Figure 5.20 and the WS-CDL iteration that represents it would be rewritten as shown in Figure 5.6.1.

5.6.2 Travel Reservation System

System Definition

This system consists of three participants: a Traveler, a Travel Agent and an Airline Reservation System, whose behavior is as follows:

A Traveler is planning on taking a trip. Once he has decided the concrete trip he wants to make he submits it to a Travel Agent by means of his local Web Service software (*Order Trip*). The Travel Agent selects the best itinerary according to the criteria established by the Traveler. For each leg of this itinerary, the Travel Agent asks the Airline Reservation System to verify the availability of seats (*Verify Seats Availability*).

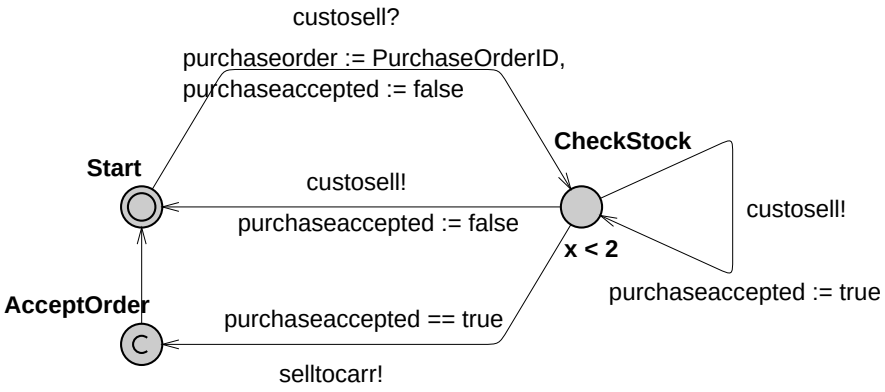


Figure 5.20: Corrected Seller automaton

```
<interaction name="PickUpProductPO" channelVariable="tns:deliver-channel"
             operation="PickUpPurchaseOrder" align="true" initiate="true">
  <participate relationshipType="tns:CustIntSellCarrRS"
               fromRole="tns:Seller" toRole="tns:Carrier"/>
  <exchange name="request"
            informationType="tns:purchaseOrderType" action="request">
    <send variable="cdl:getVariable("tns:purchaseOrder", "", "")" />
    <receive variable="cdl:getVariable("tns:purchaseOrder", "", "")"
             recordReference="record-the-channel-info" />
  </exchange>
  <timeout  time-to-complete=
             "cdl:minor(cdl:getVariable("tns:Clock1","",""),"02:00")"/>?
</interaction>
```

Figure 5.21: Corrected interaction

Thus, the Traveler has the choice of accepting or rejecting the proposed itinerary, and he can also decide not to take the trip at all.

- In the case where he rejects the proposed itinerary, he may submit the modifications (*Change Itinerary*), and wait for a new proposal from the Travel Agent.
- In the case where he decides not to take the trip, he informs the Travel Agent (*Cancel Itinerary*) and the process ends.
- In the case where he decides to accept the proposed itinerary (*Reserve Tickets*), he will provide the Travel Agent with his Credit Card information in order to properly book the itinerary.

Once the Traveler has accepted the proposed itinerary, the Travel Agent connects with the Airline Reservation System in order to reserve the seats (*Reserve Seats*). However, it may occur that at that moment no seat is available for a particular leg of the trip, because some time has elapsed from the moment in which the availability check was made. In that case the Travel Agent is informed by the Airline Reservation System of that situation (*No seats*), and the Travel Agent informs the Traveler that the itinerary is not possible (*Notify of Cancelation*).

Once the reservation is made the Travel Agent informs the Traveler (*Seats Reserved*). However, this reservation is only valid for a period of one day, which means that if a final confirmation has not been received in that period, the seats are unreserved and the Travel Agent is informed. Thus, the Traveler can now either finalize the reservation or cancel it. If he confirms the reservation (*Book Tickets*), the Travel Agent asks the Airline Reservation System to finally book the seats (*Book Seats*).

According to the previous description, the high level flow of the messages exchanged within the global process (which is called *PlanAndBookTrip*) is that shown in Fig. 5.22.

System Analysis

In this phase we identify the main requirements that the system must fulfil. Figure 5.23 depicts the goal model for this study case where the requirements

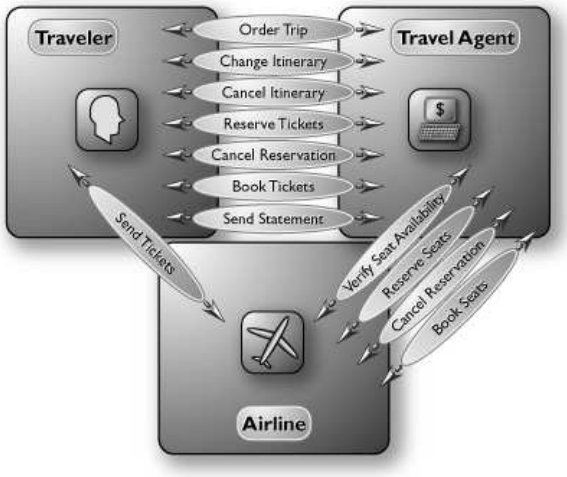


Figure 5.22: Flow of the messages exchanged.

are represented. In this figure, the root requisite is specified as *Correct Travel Reservation System* and is refined with and AND-refinement that consists of three subgoals, *System Behaves Properly*, *No Deadlocks* and *Customer Satisfaction*. The second goal is defined as a "maintenance" goal and is specified as: $A[]$ nodeadlocks.

The first subgoal, System Behaves Properly, has a "maintenance" character that is inherited by its subgoals in case they do not have any characters themselves. This subgoal is refined by an AND-refinement that consists of two subgoals which check the reservation and booking subsystem. The reservation subsystem goal, *Correct Reservation System*, captures the requirements of *Cancel On Demand* and *Correct Seat Reservation*. The booking subsystem goal, *Correct Booking System*, captures the requirements of *Booking Period is 24 hours* and *Seats Booking When Payment*.

The *Cancel On Demand* subgoal has the character of an "inevitably" goal. This goal controls the case in which the TravelAgent always cancels the reservation on traveler's demand and is specified as follows:

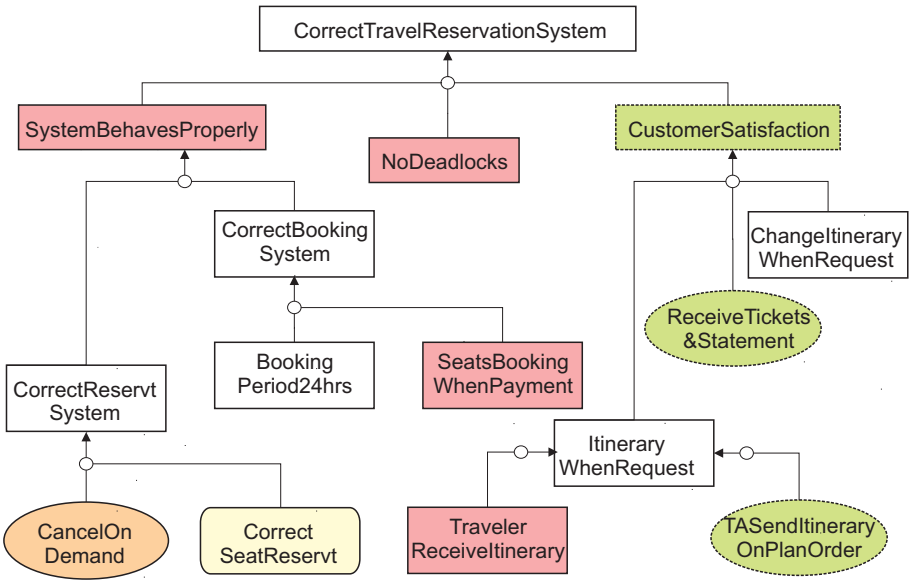


Figure 5.23: Goal Model used for the Travel Reservation System

$$\forall\Diamond Traveler.CancelReservation \rightarrow \\ (TravelAgent.CancelReservtRcv \wedge \\ Airline.PerformCancel \wedge \\ Airline.Clockx < 24) \tag{5.9}$$

The *Correct Seat Reservation* (CorrectSeatReservt) subgoal has the temporal character of "possible always". This requisite is expressed by: "If the traveler performs a reservation order, then the travel agent will send the order to the Airline and the Airline will perform the seat reservation", and is specified as follows:

$$\exists\Diamond Traveler.PerformReservation \rightarrow \\ (TravelAgent.SendReservtOrder \wedge \\ Airline.ReserveSeats \wedge \\ Airline.Clockx == 0) \tag{5.10}$$

The 5.9 and 5.10 requirements check the temporal restrictions over the clock, $Airline.Clockx$. Requirement 5.9 checks the restriction that cancelation is performed within an interval of 24 hours since the reservation has been performed, and 5.10 checks that the clock must be reset when the reservation has been accepted by the Airline.

The *Booking Period is 24 hours* subgoal has not any character, although it inherits the "maintenance" character from *System Behaves Properly*. This requirement establishes the interval in which the booking process is available. This time interval is set to 24 hours.

$$\forall\Box(TravelAgent.Booking \wedge \\ Airline.ReceiveBoking \wedge \\ Airline.ClockX <= 24) \tag{5.11}$$

The *Seats Booking When Payment* goal has the "maintenance" character and determines that the booking is carried out only if the traveler has made the payment.

$$\forall\Box Traveler.PerformPayment \rightarrow \\ Airline.BookSeats \wedge Airline.ClockX <= 24 \tag{5.12}$$

The second subgoal, *Customer Satisfaction*, is refined by an AND-refinement in three subgoals *Itinerary When Request*, *Change Itinerary When Request* and *Receive Tickets & Statement*. These three refined subgoals inherit the "achieve" character from the previous *Customer Satisfaction* requirement.

The subgoal *Itinerary When Request* is refined by an OR-refined in two subgoals *Traveler Receive Itinerary* and *TA Send Itinerary On Plan Order* (Traveler Agent Send Itinerary When Receive the Plan Order). Remember that it is enough that the system fulfils one of the subgoals to satisfy the root goal when we use the OR-refinement.

The *Traveler Receive Itinerary* leaf goal has a "maintain" character that defines the requirement that the traveler receives the itinerary when he asks for a plan order, and is specified as follows:

$$\forall\Box Traveler.PlanOrder \rightarrow TravelAgent.SendItinerary \tag{5.13}$$

The *Travel Agent Sends Itinerary On Plan Order* leaf goal has the unbounded response character. This subgoal captures the requirement that the travel agent will send the itinerary when receiving the travel request for a Plan Order and, is specified as follows:

$$Traveler.PlanOrder \longrightarrow TravelAgent.SendItinerary \tag{5.14}$$

The *Change Itinerary When Request* leaf goal has not any character; however, this requirement inherits the “achieve" character from the previous *Customer Satisfaction* goal. This goal represents the case in which the traveler may change the itinerary, and is specified as follows:

$$\exists\Diamond Traveler.ChangeItinerary \rightarrow TravelAgent.PerformChange \tag{5.15}$$

The *Receive Tickets & Statement* leaf goal has the unbounded response character. This requisite defines the obligation for the Airline and Travel Agent to send the Tickets and Statement respectively to the traveler once he has performed the booking and payment. This goal is specified as follows:

$$Traveler.PaymentPerform \longrightarrow (Traveler.Finish \wedge Airline.SnddTckt \wedge TravelAgent.SenddSttment) \tag{5.16}$$

To summarize this goal model, notice that the properties 5.9, 5.10, 5.11, 5.12, 5.15 and 5.16 must be fulfilled by the system, and properties 5.13 and 5.14 are exclusive, just one of them must be fulfilled.

System Design

In this phase the main goal is to describe the behaviors and constraints that we have specified previously. Entities, Behaviors, Communications, restrictions,

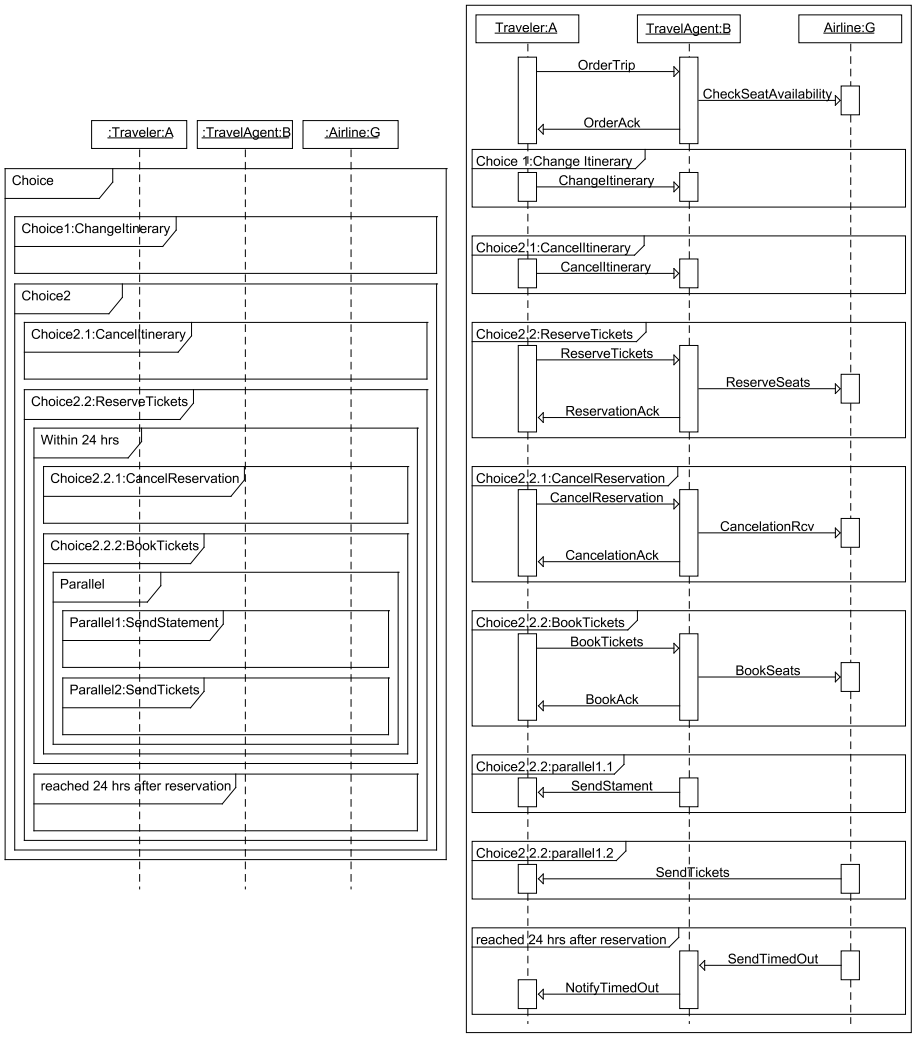


Figure 5.24: Sequence Diagram for the Travel Reservation System

etc. are derived from the analysis and modeled as UML sequence and activity diagrams.

The sequence diagram models the communication process that is exchanged between different entities and objects. Figure 5.24 shows the sequence diagram of this case study that depicts the interaction between the different parties. This diagram depicts three objects, a traveler, a travel agent and an airline.

The diagram is decomposed into two diagrams in order to get better readability. Left-hand side diagram shows the general structure by using frames and the right-hand side diagram shows the messages and the internal frames that contain the messages.

The **left-hand side diagram** consists of twelve nested frames. These frames are labeled with choices, time constraints and parallelism sequences, which capture the different behaviors of the case study.

The first frame is a choice and has two internal options “Choice 1: ChangeItinerary" and “Choice 2". This second choice also consists of two sub-options “Choice 2.1: CancelItineray" and “Choice2.2: ReserveTickets". The Choice 2.2 frame is an if-else control structure whose conditional behavior is determined by two temporal constraints “Within 24 hrs since reservation" and “Reached 24 hrs after reservation". The “if clause" consists of two choices “Choice 2.2.1: CancelReservation" and “Choice 2.2.2: BookTicket". The choice 2.2.2 consists of three messages that are followed by a parallel frame that is divided into two frames “Parallel 1: SendStatement" and “Parallel 2: SendTickets" whose sequences are performed in parallel.

The **right-hand side diagram** depicts the messages exchange and the internal frames, gathered from the structural sequence in the left-hand side diagram.

This sequence starts when the traveler orders a trip to the travel agent, who contacts with the airline to check the “Seat Availability" and sends several possible itineraries as response to the order. This sentence corresponds to the three first messages exchanged between the three parties.

Once the traveler knows the possible itineraries, he could change the default itinerary, which corresponds with the shortest one. This possibility is depicted in the diagram by the “Choice 1: ChangeItinerary" frame. On the other hand, the traveler could cancel the itinerary by performing the choice “Choice 2.1: CancelItinerary" or reserve tickets by using the choice: “Choice 2.2: ReserveTickets".

The reservation process represented by *Choice 2.2* establishes a temporal constraint: “The reservation is only available for one day". Thus, this constraint is depicted in the figure by using two frames that are labeled with “within 24

hrs" and “reached 24 hrs after reservation", which correspond to the if and else clauses respectively. The first frame depicts the message sequence when the reservation is available and the second one depicts the message sequence when the reservation has expired.

Within the reservation interval, the traveler has two options “cancelreservation" or “booktickets". This possibility is depicted by two choices: *Choice 2.2.1* or *Choice 2.2.2*. The *Choice 2.2.1* cancels the reservation whereas *Choice 2.2.2* performs the booking process. The booking process consists of a sequence of three messages, which perform the booking of tickets and seats, following by a parallel frame structure divided into two sub-frames *Parallel 1.1* and *Parallel 1.2*. This parallel structure depicts the send-receive message sequence of the statement and tickets.

If the traveler does not perform *Choice 2.2.1* or *Choice 2.2.2* within the reservation interval then a time-out notification is sent to the traveler. This message sequence is depicted by the frame *reached 24 hrs after reservation*.

The activity diagrams represents the internal behavior of each participant. Figures 5.25, 5.26 and 5.27 depict the activity diagrams of the traveler, the travel agent and the airline parties respectively. These figures show the tasks, decisions, messages, clocks, synchronizations and restrictions that the parties perform.

Figure 5.25 depicts the activity diagram of a traveler. This diagrams starts when the traveler orders a travel and the travel agent sends the acknowledgment message with the proposed itinerary. The traveler then checks the itinerary and decides between performing a change, a cancelation or a reservation. If the traveler finally performs the reservation and there are available seats for the itinerary, the airline provides an interval of one day to perform the booking. Within this interval the traveler must decide between canceling or booking the reservation. If he decides to perform the reservation, then he receives the tickets and the statement. On the other hand if the interval has expired a cancel notification is received.

Figure 5.26 depicts the activity diagram of a travel agent. The travel agent acts as a mediator between the traveler and the airline. The activity begins when the travel agent receives an order from a traveler, then the travel agent

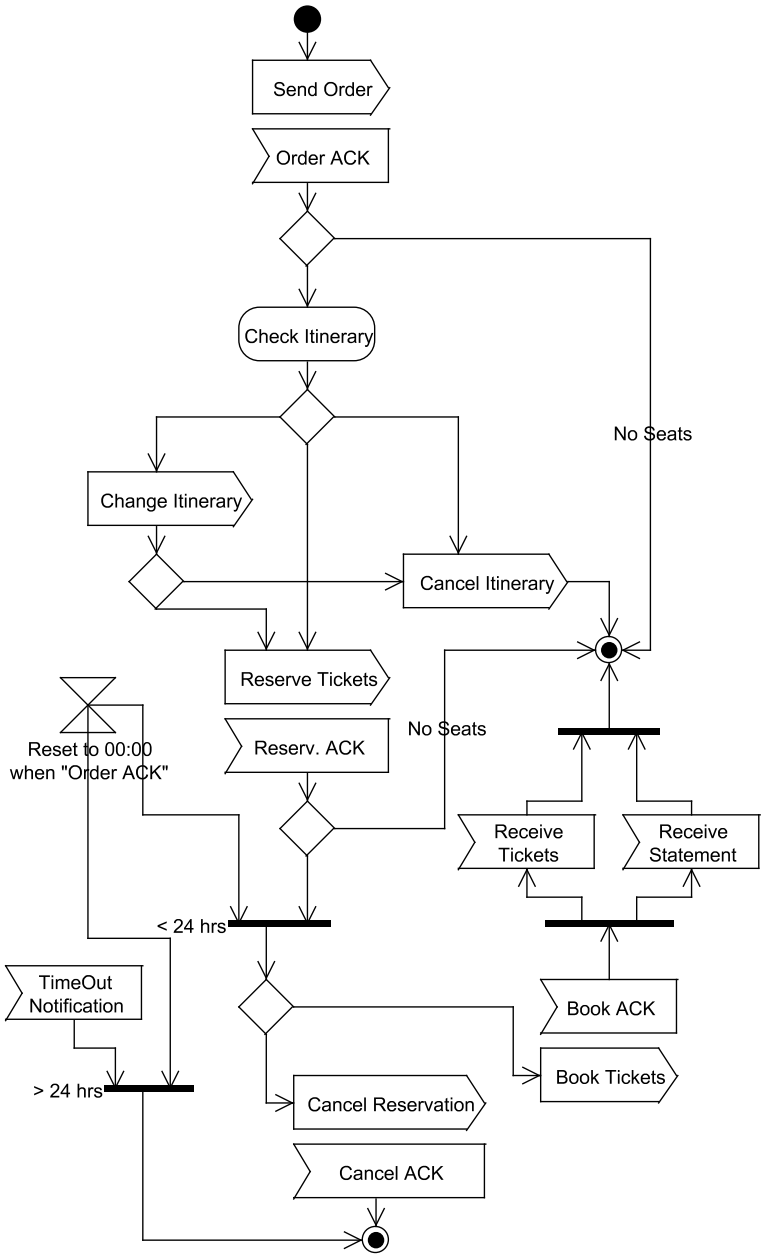


Figure 5.25: Activity diagram for a traveler process

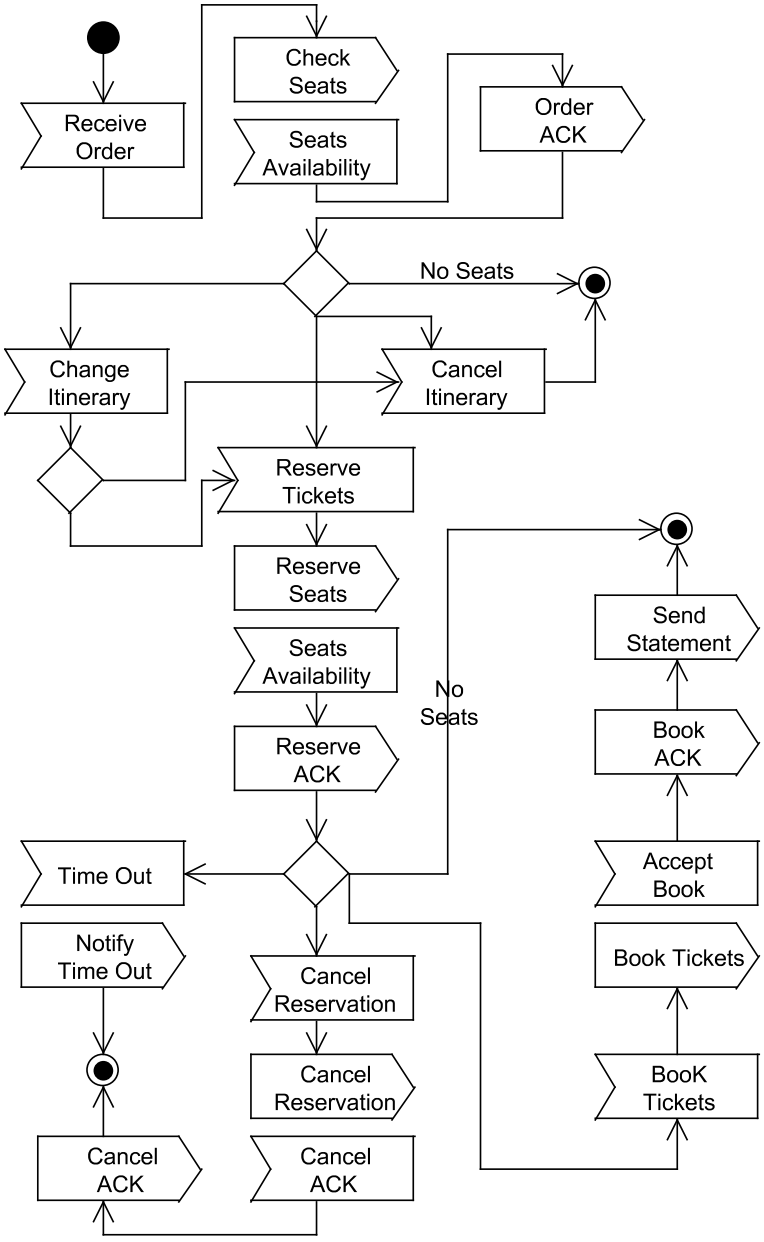


Figure 5.26: Activity diagram for a travel agent process

consults the airline about the seat availability. If no seats are available then the process ends. On the other hand, if there are available seats, then the travel agent waits for a message from the traveler. This message could be of one of three types: a cancel message, a change message or a reserve message. If the traveler has decided to reserve the itinerary (a reserve message has been received), the travel agent consults the airline about the seats availability for the chosen itinerary. If there are available seats, the travel agent confirms the availability to the traveler with a reservation acknowledgment. Now, the travel agent must wait for a response from the traveler or a notification of time-out from the airline. The traveler can reply by canceling the reservation or by booking it. If the traveler performs the booking, then the travel agent sends the statement to the traveler.

Figure 5.27 depicts the activity diagram for an airline. This process starts when a Check Seat message arrives. This message triggers the verify seat availability activity: the airline verifies the seats availability for the incoming request. After this activity has finished, the airline sends the results to the travel agent and, if there are no available seats, the process ends. On the other hand, the airline waits for a reservation seat message or a cancel itinerary message. If a reservation seat message is received, then the airline performs the reserve seat activity. However, it is posible that no seat is available at that time. Thus, if there are not available seats, the airline informs the travel agent about the impossibility of reserving these seats by sending the reserve ACK message and the process ends. If seats are available then the reserve ACK message consists of a positive notification and the clock is reset in order to activate the three synchronizations. After it three cases could occur: the first one is, no news are received from the travel agent in 24 hours, then the left synchronization is activated and a notification of time-out is sent to the travel agent. The second case occurs when a cancel reservation message is received within 24 hours, and then, the process ends by sending a cancelation accepted message is generated. The third case occurs when a book seats message is received within 24 hours, and then, the airline performs the book seat activity and sends two messages, seats booked to the travel agent and send tickets to the traveler.

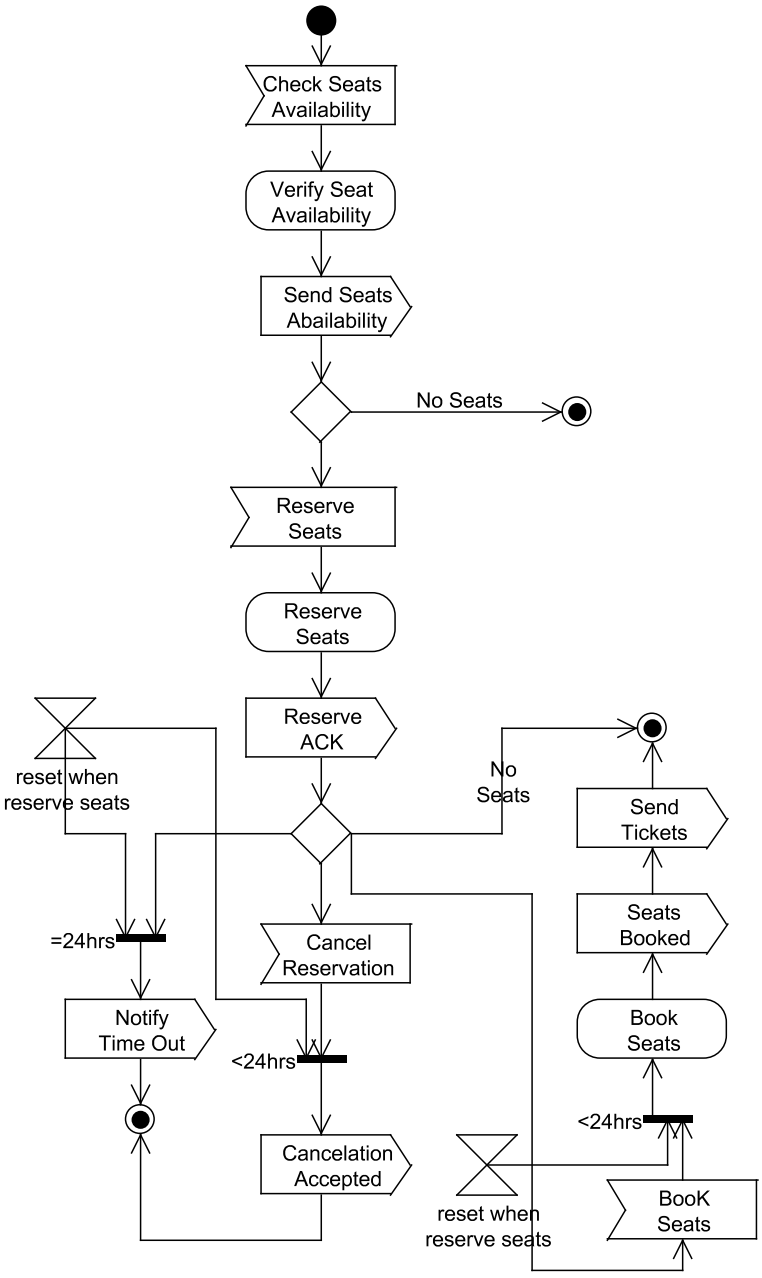


Figure 5.27: Activity diagram for an airline process

```
<interaction
 name="reservation&booking" align="true"
 channelVariable="travelAgentAirlineChannel"
 operation="reservation&booking" initiate="true" >
 <participate
   relationshipType="TravelAgentAirline"
   fromRole="TravelAgent" toRole="Airline" />
 <exchange name="reservation"
  informationType="reservation" action="request" >
  <send variable="tns:reservationOrderID"
   causeException="true" />
  <receive variable="tns:reservationAckID"
   causeException="true" />
 </exchange>
 <timeout  time-to-complete="24:00" />
 <record  name="bookingTimeout"
  when="timeout" causeException="true" />
  <source variable=
    "AL:getVariable('tns:resOC', '', '')" />
  <target variable=
    "TA:getVariable('tns:resOC', '', '')" />
   </record>
</interaction>
```

Figure 5.28: Part of WS-CDL specification

WSCDL Documents

Figure 5.28 presents a detailed piece of the WS-CDL document describing the example that we have used to obtain the translation into timed automata. Following the guidelines described above we have obtained in this case three timed automata: the traveler, the travel agent and the airline company. These automata are shown in Figures 5.29, 5.30 and 5.31.

Notice the use of the clock x in the timed automaton corresponding to the airline reservation system, which is used to control when the reservation expires. This clock is initialized when the action *reserved_seat* is carried out.

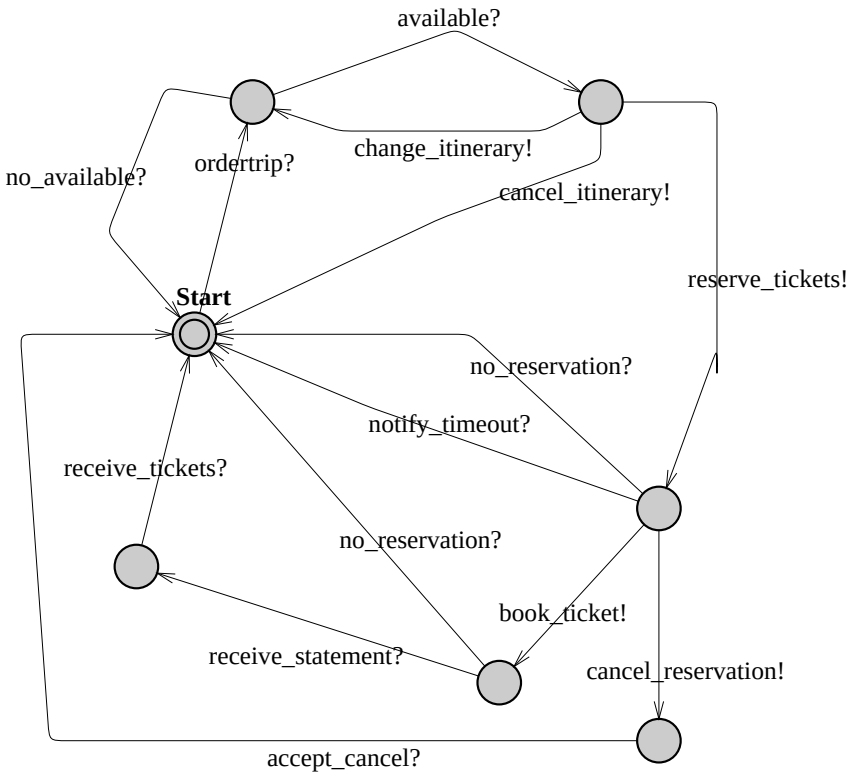


Figure 5.29: Timed automata for traveler.

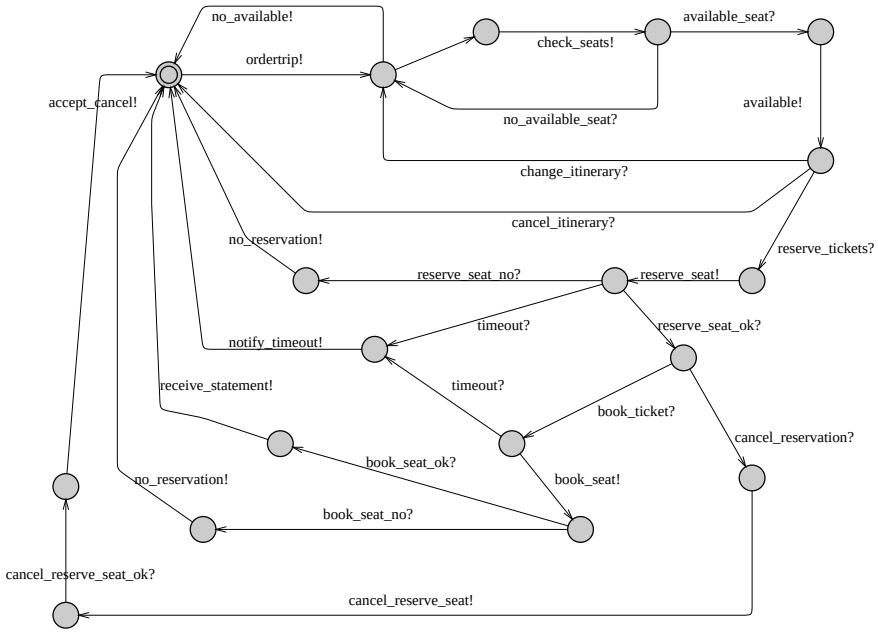


Figure 5.30: Timed automata for Travel agent Web Service.

Simulation and Verification

In the simulation we can check whether the model keeps the system behavior or not. This can partially be done by means of simulations. These are made by choosing different transitions and delays along the system evolution. At any moment during the simulation, you can see the variable values and the enabled transitions. Thus, you can choose the transition that you want to execute. Nevertheless, you can also select the random execution of transitions, and thus, the system evolves by executing transitions and delays which are selected randomly. We have some other options in the Simulator. For instance, you can save simulation traces that can later be used to recover an specific execution trace. Actually, the simulation is quite flexible at this point, and you can move back or forward in the sequence. Having made a number of simulations, we have concluded that the system design satisfies the expected behavior in terms of the message flow between the parties.

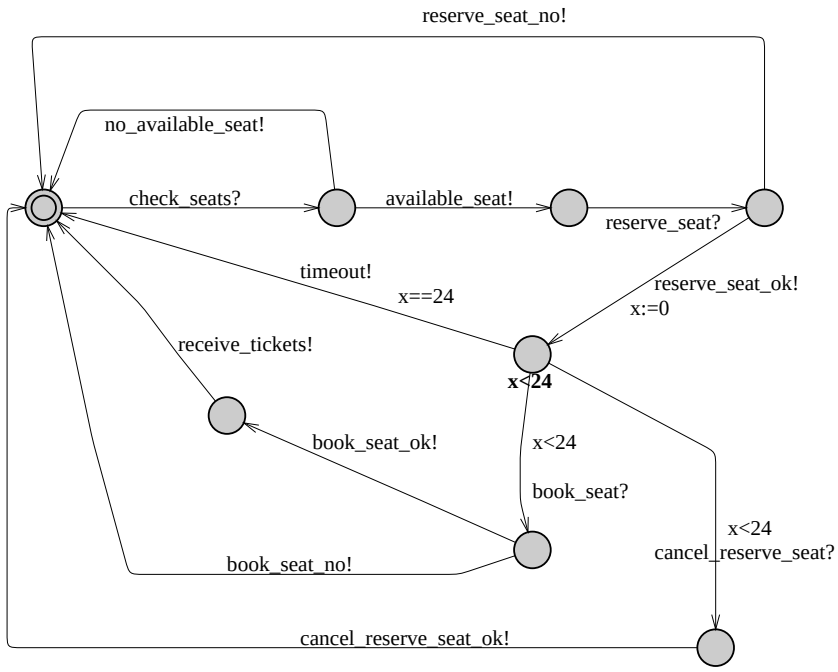


Figure 5.31: Timed automata for airline Reservation System.

Then, with respect to our study case, our main goal in the validation-verification phase is to check the correctness of the goal model developed in the system analysis (Figure 5.23). According to that model, properties 5.9, 5.10, 5.11, 5.12, 5.15 and 5.16 have been proved to be fulfilled, while properties 5.13 and 5.14 are equivalent properties, i.e., it is enough, if only one of them is fulfilled.

CHAPTER 6

Conclusions & Future Research

6.1 Conclusions

In the second chapter, we have reviewed the most important formal methods and techniques developed for the analysis and verification of software and hardware systems. Among these techniques and methods, we have distinguished “Model Checking" from the others because this technique has several advantages that make it suitable for our purposes. However, we should not forget that also has some drawbacks. The main advantages that have leaded us to select it are the following:

- It is a general verification approach that it is applicable to a wide range of applications.
- It is not vulnerable to the error occurrence, i.e. other techniques such as simulation and Testing check only the most probable defects, while model checking checks all possible defects.
- It provides “diagnostic information" in order to locate where an error is.

- It can easily integrated in existing development cycles.
- Several tools support it.

We have applied the “Model Checking" to two different types of systems: Real-Time Systems and Probabilistic Systems. Furthermore, we have also studied a technique to analyze at the same time systems with both characteristics, temporal and probabilistic. The analysis and verification of Real-Time Systems have been performed by the UPPAAL tool, while the analysis of probabilistic systems have been performed by the P_UPPAAL tool.

The UPPAAL tool uses a network of Timed Automata to model real time systems, a subset of the temporal logic TCTL for specifying the properties that the system must satisfy and an efficient verifier that checks the properties over the model.

The P_UPPAAL tool uses a probabilistic transition system to model the Probabilistic Systems, reachability properties that determine the probability to reach a state from a previous state and the RAPTURE verifier engine that makes successive refinement to check whether the reachability property is true or not.

Furthermore, by using an integral semantic with P_UPPAAL it is possible to analyze Probabilistic Real-Time Systems, but we must take into account that this integral semantic is not dense. Thus, this technique is only suitable for those systems where the temporal constraints consist of no restrictive restrictions (i.e. do not consist of “<" or “>", but “==", “<=" or “>=").

These techniques have been applied to the most emergent area in the recent decades, Internet. The use of Internet has extended from all around the world and it has led to a depth change in the manner in which our society communicates. However, the Internet growing is threatened by two serious problems, security and correctness.

Security problems one of the main goals for e-business systems. These aspects have been studied in depth by many authors, defining and analysing some security protocols on the Internet. The main security holes studied by them are authentication and cryptographic problems, while time aspects have not received so much attention. Thus, we have performed, in chapter 3, the analysis and verification of time aspects of two security protocols, the TLS handshake

protocol and the SET protocol. From these studies we have concluded that it is necessary the analysis and verification of time aspects, not just as a secondary goal, but as a primary goal of security protocols.

Correctness problems concern in general to all software systems, however they have a remarkable importance in systems where coordination is crucial, i.e. systems consisted of several parties that are not homogeneous, such as Web Services. The correctness analysis of Web Services has led towards the highest layers of the protocol stack, choreographies and orchestrations.

We have introduced in chapter 5 a methodology for the generation of “correct" Web Services by using formal techniques, but firstly, we have introduced what the definition of a “correct" Web Service is. This definition determines that a “correct" Web Service should satisfy some properties inherent to the system and other properties more general that are inherent to any concurrent and real time system.

This methodology proposes a top down design that consists of analysis, design, verification and generation. The analysis specifies the properties that the system must satisfy. We have used a goal model that adequately captures time requirements. This goal model is an extension of KAOS goal model that uses a tree structure, where the properties are defined from general and strategic goals at the roots, and concrete goals at the leaves. Property leaves are specified in the temporal logic used by UPPAAL.

The design is based on UML sequence and activity diagrams. We have used an extended version of these diagrams to properly capture the time characteristics and constraints. These diagrams are translated into Web Service choreographies that describe the coordination of several Web Services.

Verification checks the properties gathered in the analysis phase. This verification process uses UPPAAL as a verification engine. This tool checks timed automata networks, so it is necessary as a previous step to translate the choreographies into this type of model (by using XSLT rules). Once this transformation is performed, it is possible to check whether the system holds the gathered properties or not. If the model does not satisfy a concrete property, then it is possible to fix the problem by returning to a previous phase.

In order to illustrate this methodology, we have used two particular case studies: an Internet purchase and a travel reservation system.

6.2 Current Work

The methodology that we have introduced in the previous chapter of this dissertation can be improved, by adding some other techniques that are widely used to describe Web Services. Then, our current work focuses on the diagram show in Fig. 6.1, in which the relationship between the different elements of this methodology is depicted.

The phase 1, the system analysis, is represented with a pink box label with the goal-oriented framework KAOS that performs the requirement engineering. The phase 2, the system design, is represented by using an orange box that is labeled with the UML diagrams sequence and activity, which will be translated into WS-CDL choreography.

The phase 3, the system modeling and verification, transforms the choreographies written in WS-CDL into timed automata by using some XSLT rules. The phase 4, the generation of Web Services, is implemented by the translation to WS-BPEL, which is based on XSLT transformation rules too.

Thus, the process for the generation of correct Web Services can now by summarized in the following steps:

1. The requirements that the system must fulfil are captured by using KAOS.

2. UML sequence and activity diagrams are constructed starting from the requirements.

3. UML diagrams are translated into the choreography layer (WS-CDL).

4. WS-CDL specification documents are translated into timed automata with XSLT rules.

5. The generated timed automata are verified with UPPAAL. The properties to check are those that we have obtained in the first step. If failures are detected, then them must be corrected, and return to the second step.

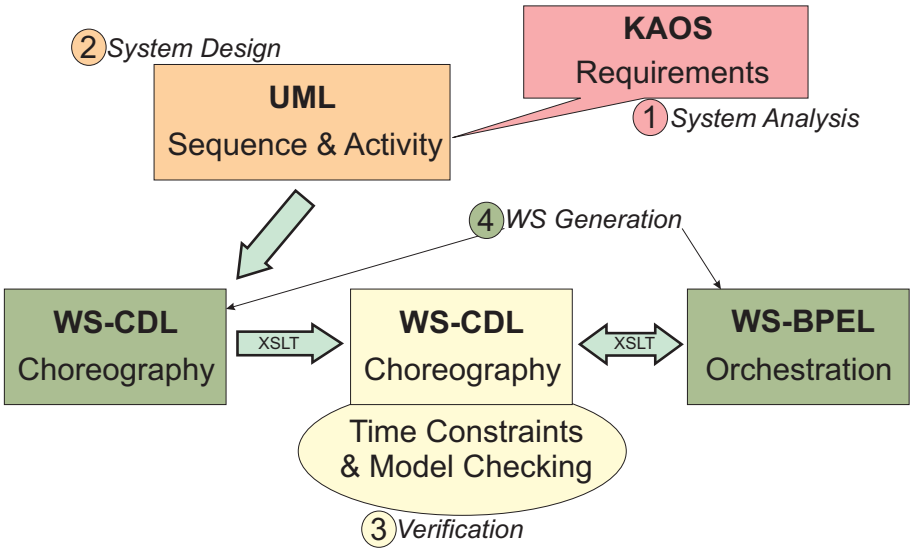


Figure 6.1: Methodology diagram

6. Timed automata are transformed into WS-BPEL orchestration specifications by using XSLT rules and these orchestrations are enriched with each concrete behavior.

7. The enriched WS-BPEL orchestration are translated into timed automata in order to recheck the properties. Again, if some failure is detected, we go back to the fifth step in order to fix the problem.

In this dissertation the steps 1,2,4 and 5 have been covered, while steps 3, 6 and 7 are "work in progress". Figure 6.2 depicts the transformation steps 4 and 6 and Figure 6.3 depicts a basic example of these steps. This example consists of two parties, A and B. These parties interact by synchronizing in a loop, which can repeat the interaction, if the "x" clock has not reach 5 seconds. Once the synchronization has been performed, the B party increments and stores the global "i" variable. Notice that, in order to clarify this basic example, we have simplified the notation of the WS-CDL and WS-BPEL documents.

In the WS-CDL specification the parties are described as roles, <roleType name="A"> and <roleType name="B">. The integer variable i and the clock x

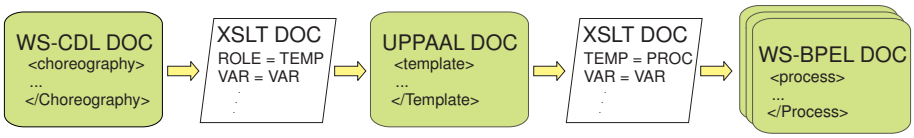


Figure 6.2: Automatic translations by applying XSLT transformation rules.

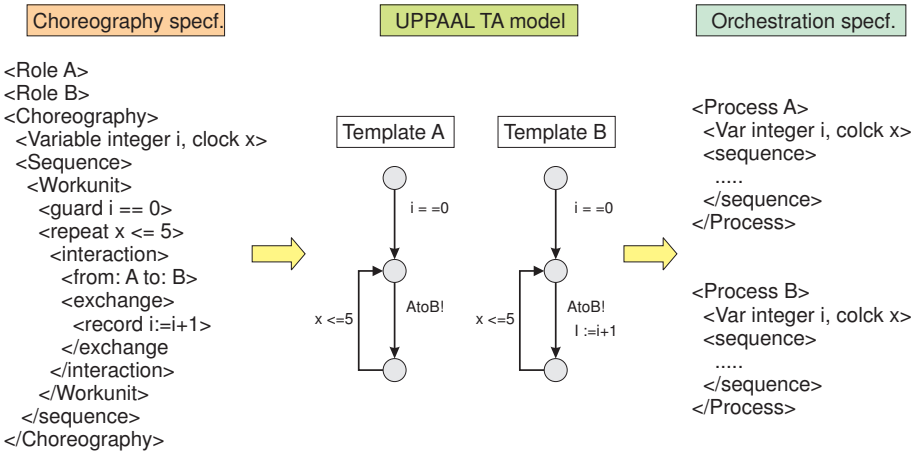


Figure 6.3: Translation simple example.

as variables and the loop is specified as a workunit with an interaction between A and B that could be repeated by means of the repeat clause of this workunit.

Once the XSLT rules has been applied to this WS-CDL specification we obtain a timed automata that consists of two templates, A and B. Both templates are symmetric and consist of an initial state with an outgoing transitions that can be performed when the “i" integer variable is zero. furthermore, once performed this action, we have a loop that is repeated while the “x" clock does not reach more than 5 seconds. This loop synchronizes both automata and B automaton increments and stores “i".

On the other hand, in the WS-BPEL specifications obtained by applying other XSLT rules (work in progress), the UPPAAL templates are described as processes, <process name=“A"> and <process name=“B">. The integer variable i and the clock x are described as local variables and the loop is specified

as a if-then-else structure inside a flow structure with a message between A and B that could be repeated by means of the condition clause of the if-then-else structure.

6.3 Future Research

Our future work is will focus on three different objectives. The first objective is the study of efficient Model Checking techniques applied to Probabilistic Real Time Systems. The second objective is to standardized the time analysis of security protocols, and the third, will consist in the development of tools and techniques for improving our proposal for generating “correct" Web Services.

The first objective is motivated by the necessity of covering not only discrete time systems, but also systems with probabilistic and real-time characteristics (by using a dense semantic). In order to achieve this first objective, we will join the present Model Checking techniques for Real Time Systems with the analysis techniques for probabilistic systems. Thus, this work should combine the efficient techniques of symbolic model checking with the refinement techniques for computing extremum probabilities.

The second objective is defined around the idea of getting closer the timed automata with the languages used for the verification and analysis of security protocols. Timed automata describe the concrete behaviors of systems, while the languages for modeling security protocols are more abstracts. Thus, we will work in the generation of standardized security protocol behaviors captured by means of timed automata, i.e., we will map authentication and cryptographic tasks with theirs corresponding timed automata behaviors. This work will allow us to properly study the security risks form several view points, authentication, cryptography, passage of time, etc. Furthermore, we are interested in applying these techniques to the Web Services Security Protocol family.

The third objective in which we are currently working is the development and improvement of techniques and tools for the proposal of generating “correct" Web Services. This objective is also divided into three sub-objectives, which are improvements in the analysis, design and generation phases.

concerning the analysis phase, we are developing an algorithm to check

the consistence of the gathered properties, i.e. we want to check whether the gathered properties have any contradictions or not.

In design aspects we will consider RT-extensions of the UML sequence and activity diagrams (UML profile for Real Time Systems), with the intention to describe Web Services oriented to real-time applications.

In the generation phase the main problem concerns to the orchestration semantics. There is not a clear semantics for the Business Process Execution Language (WS-BPEL). Thus, our intention is to use or define a standardized semantics for it.

6.4 Outstanding Contributions and International Collaborations

Contribution in articles of journals, conferences and technical reports in chronological order, and a brief description of them follows:

- Maria del Carmen Carrión, Gregorio Díaz and Blanca Caminero *Performance Issues of Deterministic and Adaptative Ghost-Packet Routers*, Proceedings of the 2001 International Conference on Parallel Processing, 3—7 September, IEEE Computer Society Press.

- María del Carmen Carrión, Gregorio Díaz and Blanca Caminero, *A low—cost Router Design for k—ary n—cube Networks*, Technical Report de la UCLM, 2002, ref "DIAB—02—02—30".

- Gregorio Díaz , Fernando Cuartero and Kim G. Larsen, *P_UPPAL a tool for capturing the probabilistic behavior of UPPAAL models*, Technical Report de la UCLM, 2002, ref "DIAB—02—01—33".

- Gregorio Díaz , Fernando Cuartero and Diego Cazorla, *P_UPPAL una herramienta de modelado, validación y verificación de Sistemas Concurrentes Probabilísticos*, submitted to XI Jornadas de Concurrencia 2003.

- Gregorio Díaz, Diego Cazorla, Fernando Pelayo, Fernando Cuartero y Valentín Valero, *Verifying and Capturing Probabilistic Behavior of Real-time Systems*, proceedings of the UK Performance Engineering Workshop,

UKPEW 2003.

- Gregorio Díaz, Fernando Cuartero, Valentín Valero and Fernando L. Pelayo, *Automatic Verification of the TLS Handshake Protocol*, In proceedings of the 2004 ACM Symposium on Applied Computing, ACM press.

- Gregorio Díaz, Kim G. Larsen, Juan José Pardo, Fernando Cuartero and Valentín Valero, *An approach to handle Real Time and Probabilistic behaviors in e-commerce: Validating the SET Protocol*, In proceedings of the 2005 ACM Symposium on Applied Computing, ACM press.

- Gregorio Díaz, Juan José Pardo, Maria Emilia Cambronero, Valentín Valero and Fernando Cuartero, *Verification of Web Services with Timed Automata*, In proceedings of First International Workshop on Automated Specification and Verification of Web Sites 2005, Springer Verlag´s Electronics Notes in Theoretical Computer Science series, volume 2380.

- Gregorio Díaz, Juan José Pardo, Maria Emilia Cambronero, Valentín Valero and Fernando Cuartero, *Automatic Translation of WS-CDL Choreographies to Timed Automata*, In proceedings of the international Workshop on Web Services and Formal Methods 2005, Springer Verlag's Lecture Notes in Computer Science series, volume 3670.

- Llanos Tobarra, Diego Cazorla, Fernando Cuartero and Gregorio Díaz, *Application of Formal Methods to the Analysis of Web Services Security*, In proceedings of the international Workshop on Web Services and Formal Methods 2005, Springer Verlag's Lecture Notes in Computer Science series, volume 3670.

- Gregorio Díaz, M. Emilia Cambronero, Juan J. Pardo , Valentín Valero and Fernando Cuartero, *Automatic generation of Correct Web Services Choreographies and Orchestrations with Model Checking Techniques*, In proceedings of the International Conference on Internet and Web Applications and Services ICIW'06.

International Collaborations and stays at international universities and research groups:

- Basic Research In Computer Science at Aalborg University - Denmark, for 6 months in 2002.
- Center for Indlejrede Software Systemer at Aalborg University - Denmark, for 5 months in 2004.

APPENDIX A

XSLT Transformations

XML StyleSheets Language for Transformation (XSLT or XSL Transformation) is a standard of W3C that presents rules for the transformation between different XML document formats or even from XML formats into other formats. The style sheets perform the transformations by using template rules. These rules together the original document are transformed by a XSLT parser into a result document.

We describe now the style sheet and its template rules used in the transformations.

The XSL sheet begins with the following tags:

$<?xmlversion =' 1.0'? >< xsl : stylesheetversion =$ "1.0"$xmlns : xsl = http : //www.w3.org/1999/XSL/Transform >$ The first tag shows the XML version and the second consists of two attributes, the version and the name space for XSLT.

A.1 Preamble

Once it is defined we must start by defining the preamble.

Intended result:

```
<?xml version="1.0"?>
<DOCTYPE nta PUBLIC "-//Uppaal Team//DTD Flat System 1.0//EN"
"http://www.docs.uu.se/docs/rtmv/uppaal/xml/flat-1_0.dtd">
```

Rule:

```
<xsl:output method="xml"
doctype-public="-//Uppaal Team// DTD Flat System 1.0//EN"
doctype-system=
http://www.docs.uu.se/docs/rmtv/uppaal/xml/flat-1_0.dtd>
```

The XSLT is structured with several templates that are defined with the tag:

$< xsl : templatematch = "templatename" >$

In each XSLT document, there is a root template that begins with $< xsl : templatematch = "/" >$ and ends with $< /xsl : template >$. These templates consist of the transformation rules to be applied and they can also use other templates. To invoke other templates, we use the tag $< xsl : apply - templatesselect = "//templatename"/ >$. Once this tag appears, the defined rules for this template are applied.

The following rule is as follows:

Intended result: $< nta > ... < /nta >$

Literal rule: $< nta > ... < /nta >$

A.2 Declarations

To declare the global variables we will use the tags $< declaration > ... < /declaration >$.

To declare the channels for roles communication an others.

```
<xsl:text> chan </xsl:text>
<xsl:apply-templates select = "//variable">
<xsl:text>; </xsl:text>

<xsl:template match = "variable">
   <xsl:if test = "@channelType">
   <xsl:if test = "position()=1">
        <xsl:value-of select="@name"/>
   <xsl:if test = (position()\&gt;1) and
                  (position()\&lt; last())">
        <xsl:text>, <xsl:text>
        <xsl:value-of select = "@name"/>
   </xsl:if>
</xsl:template>
```

A.3 Templates

We use for the templates the tags <template> ... </template> and the rules:

```
<xsl:for-each select="//roletype"> </xsl:for-each>

<name> <xsl:value-of select="@name">  </name>
```

We will need to use XPath to search in the original document. The evaluation of a XPath expression returns four different types, node sets, boolean value, a number or a string. The syntaxis of XPath is as follows:

- brackets and parenthesis : ( ) { } [ ] .
- Present element . y parent element ..
- Attribute @, element '*' y blank '::'.

- The coma ',',
- The element name.
- Node type (comment, text, processing instruction, node).
- Operators: and, or, mod, div, , /, //, |, +, -, =, ! =, <, <=, >, >=.
- Functions name.
- Axis names (axis): ancestor, ancestor-or-self, attribute, child, descendant, descendant-or-self, following, following-sibling, namespace, parent, preceding, preceding-sibling, self.
- Literals, between quotation marks.
- Numbers and variable references (var_name).

These XPath expressions are used in the match, select and test attributes. Thus in order to find all the nodes it is needed to search by using the axis predicate. The axis predicates consist of Child (child elements), following-sibling (brother elements) and descendant (all descendants). Thus we will use the following rule to generate the nodes:

Locations:

$< locationid = "nombredelproceso" >$

Rules:

```
<xsl:element name="location">
  <xsl:attribute name="id">id<xsl:value-of select="@name"/>
  </xsl:attribute>
</xsl:element>
```

Initial node:

$< initref = "initial_node_name" / >$

Rules:

```
<xsl:element name="init">
  <xsl:attribute name="ref">id<xsl:value-of select="@name"/>
  </xsl:attribute>
</xsl:element>
```

Transitions:

Source: $< sourceref = "source\_id" / >$

Rules:

```
<xsl:element name="source">
  <xsl:attribute name="ref"> source\_id  </xsl:attribute>
</xsl:element>
```

Target: $< targetref = "target_id" / >$

Rules:

```
<xsl:element name="target">
   <xsl:attribute name="ref"> target\_id </xsl:attribute>
</xsl:element>
```

APPENDIX B

XML Documents for an Internet Purchase Process

B.1 WS-CDL Document

```
<?xml version="1.0" encoding="UTF-8"?> <package
xmlns="http://www.w3.org/2004/12/ws-chor/cdl"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     targetNamespace="http://www.oracle.com/ashwini/sample"
     xmlns:tns=http://www.oracle.com/ashwini/sample"
     name="PurchaseInternetProcessChoreography"
     version="1.0">

  <informationType name="purchaseOrder"
                   type="tns:PurchaseOrderMsg"/>
  <informationType name="purchOrderAccp"
                   type="tns:purchOrderAccp"/>
  <informationType name="NoStockAck"
                   type="tns:NoStockAck"/>
```

```
<token name="purchaseOrderID" informationType="tns:intType"/>
<tokenLocator tokenName="tns:purchaseOrderID"
              informationType="tns:purchaseOrderType"
              query="/PO/orderId"/>
<tokenLocator tokenName="tns:purchaseOrderID"
              informationType="tns:purchOrderAccp"
              query="/PO/orderId"/>
<roleType name="Customer">
  <behavior name="Cust4IntSell"
            interface="tns:CustIntSellPT"/>
  <behavior name="Cust4Carr"
            interface="tns:Cust4CarrPT"/>
</roleType>
<roleType name="InternetSeller">
  <behavior name="IntSell4Cust"
            interface="tns:IntSell4Cust"/>
  <behavior name="IntSell4Carr"
            interface="tns:IntSell4CarrPT"/>
</roleType>
<roleType name="Carrier">
  <behavior name="Carr4IntSell"
            interface="tns:Carr4IntSellPT"/>
  <behavior name="Carr4Cust"
            interface="tns:Carr4CustPT"/>
</roleType>
<relationshipType name="CustIntSellRS">
  <role type="tns:Customer" behavior="Cust4IntSell"/>
  <role type="tns:InternetSeller" behavior="IntSell4Carr"/>
  <role type="tns:Carrier" behavior="Carr4Cust"/>
</relationshipType>
<channelType name="CustomerChannel">
  <passing channel="CarrierChannel" action="request" />
  <role type="tns:Consumer"/>
  <reference>
    <token name="tns:customerRef"/>
  </reference>
```

```
  <identity>
    <token name="tns:purchaseOrderID"/>
  </identity>
</channelType>
<channelType name="InternetSellerChannel">
  <passing channel="CustomerChannel" action="request" />
  <role type="tns:Retailer" behavior="IntSell4Cust"/>
  <reference>
    <token name="tns:InternetSellerRef"/>
  </reference>
  <identity>
    <token name="tns:purchaseOrderID"/>
  </identity>
</channelType>
<channelType name="CarrierChannel">
  <passing channel="InternetSellerChannelChannel"
           action="request" />
  <role type="tns:Retailer" behavior="IntSell4Cust"/>
  <reference>
    <token name="tns:InternetSellerRef"/>
  </reference>
  <identity>
    <token name="tns:purchaseOrderID"/>
  </identity>
</channelType>

<choreography name="PurchaseInternetProcessChoreography"
              root="true">
  <relationship type="tns:CustIntSellRS"/>
  <variableDefinitions>
    <variable name="purchaseOrder"
              informationType="tns:purchaseOrderType"
              silent="true" />
    <variable name="purchOrderAccp"
              informationType="tns:purchOrderAccp"/>
    <variable name="NoStockAck"
```

```
            informationType="tns:NoStockAck"/>
  <variable name="Clock1"
            informationType="tns:Clock"/>
  <variable name="seller-channel"
            channelType="tns:InternetSellerChannel"/>
  <variable name="customer-channel"
            channelType="tns:CustomerChannel"/>
  <variable name="carrier-channel"
            channelType="tns:CarrierChannel"/>
</variableDefinitions>

<sequence>

<interaction name="createPO"
             channelVariable="tns:seller-channel"
             operation="handlePurchaseOrder" align="true"
             initiate="true">
  <participate relationshipType="tns:CustIntSellRS"
               fromRole="tns:Customer" toRole="tns:Seller"/>
  <exchange name="request"
            informationType="tns:purchaseOrderType"
            action="request">
    <send variable=
            "cdl:getVariable("tns:purchaseOrder","","")"/>
    <receive variable=
              "cdl:getVariable("tns:purchaseOrder", "", "")"
             recordReference="record-the-channel-info" />
  </exchange>
  <exchange name="response"
            informationType="purchOrderAccp"
            action="respond">
    <send variable=
           "cdl:getVariable("tns:purchaseOrderAcceted","","")"/>
    <receive variable="cdl:
            getVariable("tns:purchOrderAccp", "", "")" />
  </exchange>
```

```
  <exchange name="NoStockAckException"
            informationType="NoStockAckType"
            action="respond">
    <send variable="cdl:getVariable('tns:NoStockAck', '', '')"
            causeException="true" />
    <receive variable=
              "cdl:getVariable("tns:NoStockAck", "", "")"
            causeException="true" />
  </exchange>
  <record name="record-the-channel-info" when="after">
    <source variable="cdl:getVariable("tns:purchaseOrder, "",
                                      "PO/CustomerRef")"/>
    <target variable=
             "cdl:getVariable("tns:customer-channel","","")"/>
  </record>
  <record name="reset-clock" when="after">
    <source variable="00:00"/>
    <target variable="cdl:getVariable("tns:Clock1", "", "")"/>
  </record>
</interaction>

<interaction name="PickUpProductPO"
             channelVariable="tns:carrier-channel"
             operation="PickUpPurchaseOrder" align="true"
             initiate="true">
  <participate relationshipType="tns:CustIntSellRS"
               fromRole="tns:Seller" toRole="tns:Carrier"/>
  <exchange name="request"
            informationType="tns:purchaseOrderType"
            action="request">
    <send variable=
            "cdl:getVariable("tns:purchaseOrder", "", "")" />
    <receive variable=
              "cdl:getVariable("tns:purchaseOrder", "", "")"
             recordReference="record-the-channel-info" />
  </exchange>
```

```
      <timeout  time-to-complete=
       "cdl:minor(cdl:getVariable("tns:Clock1","",""),"02:00")"/>?
    </interaction>

    <interaction name="DeliverProductPO"
                 channelVariable="tns:customer-channel"
                 operation="DeliverProductOrder" align="true"
                 initiate="true">
      <participate relationshipType="tns:CustIntSellRS"
                   fromRole="tns:Carrier" toRole="tns:Customer"/>
      <exchange name="request"
                informationType="tns:purchaseOrderType"
                action="request">
        <send variable=
                "cdl:getVariable("tns:purchaseOrder", "", "")" />
        <receive variable=
                  "cdl:getVariable("tns:purchaseOrder", "", "")"
                 recordReference="record-the-channel-info" />
      </exchange>
      <timeout
        time-to-complete=
          "cdl:minor(cdl:getVariable("tns:Clock1", "", "")
                                                ,"24:00")"/>?
    </interaction>
    </sequence>
 </choreography>
</package>
```

B.2 WS-BPEL Document

B.2.1 Definitions

The WSDL portType offered by the service to its customers (purchaseOrderPT) is shown in the following WSDL document. Other WSDL definitions required by the business process are included in the same WSDL document for simplicity:

```
<definitions
targetNamespace="http://manufacturing.org/wsdl/purchase"
        xmlns:sns="http://manufacturing.org/xsd/purchase"
        xmlns:pos="http://manufacturing.org/wsdl/purchase"
        xmlns="http://schemas.xmlsoap.org/wsdl/">

        <types>
          <xsd:schema >
           <xsd:import
               namespace="http://manufacturing.org/xsd/purchase"
          schemalocation="http://manufacturing.org/xsd/purchase.xsd"/>
         </xsd:schema>
        </types>



        <message name="POMessage">
             <part name="customerInfo" type="sns:customerInfo"/>
             <part name="purchaseOrderID" type="sns:purchaseOrderID"/>
        </message>

        <message name="AcceptanceMessage">
             <part name="Acceptance" type="sns:Acceptance"/>
        </message>

        <message name="DeliverMessage">
             <part name="DeliverMessage" type="sns:Invoice"/>
        </message>

        <message name="orderFaultType">
              <part name="problemInfo" element=”sns:OrderFault"/>
        </message>


<!-- portType supported by the customer process -->
```

```
        <portType name="custosellPT">
             <operation name="sendPurchaseOrder">
               <output message="pos:POMessage"/>
             </operation>
             <operation name="receiveAcceptanceOrder">
               <input message="pos:AcceptanceMessage"/>
             </operation>
        </portType>

        <portType name="carrtocusPT">
             <operation name="receiveProduct">
               <input message="pos:DeliverMessage"/>
             </operation>
        </portType>


<!-- portTypes supported by the seller process -->

        <portType name="custosellPT">
             <operation name="receivePurchaseOrder">
               <input message="pos:POMessage"/>
               <fault name="cannotCompleteOrder"
                  message="pos:orderFaultType"/>
             </operation>
             <operation name="sendAcceptanceOrder">
               <output message="pos:AcceptanceMessage"/>
             </operation>
        </portType>

        <portType name="selltocarrPT">
              <operation name="sendPickUpOrder">
                <output message="pos:POMessage"/>
              </operation>
        </portType>

<!-- portType supported by the carrier process -->
```

```
        <portType name="selltocarrPT">
             <operation name="PickUpOrder">
               <input message="pos:POMessage"/>
             </operation>
        </portType>

        <portType name="carrtocusPT">
             <operation name="deliverProduct">
               <output message="pos:DeliverMessage"/>
             </operation>
        </portType>


<plnk:partnerLinkType name="Customer">
     <plnk:role name="cus4sellIP"
                portType="pos:custosellPT"/>
     <plnk:role name="cus4carrIP"
                portType="pos:carrtocusPT"/>
</plnk:partnerLinkType>

<plnk:partnerLinkType name="Seller">
     <plnk:role name="sell4cusIP"
                portType="pos:custosellPT"/>
     <plnk:role name="sell4carrIP"
                portType="pos:selltocarrPT"/>
</plnk:partnerLinkType>

<plnk:partnerLinkType name="Carrier">
     <plnk:role name="carr4cusIP"
                portType="pos:carrtocusPT"/>
     <plnk:role name="carr4sellIP"
                portType="pos:selltocarrPT"/>
</plnk:partnerLinkType>


</definitions>
```

B.2.2 The Customer Process

```
<process name="InternetPurchaseOrderCustomerProcess"
        targetNamespace="http://acme.com/ws-bp/Internetpurchase"
        xmlns="http://schemas.xmlsoap.org/ws/2004/03/business-process/"
        xmlns:lns="http://manufacturing.org/wsdl/Internetpurchase">

<documentation xml:lang="EN">A simple example of a WS-BPEL process
for handling a Internet purchase order.</documentation>

<partnerLinks>
     <partnerLink name="SellerIP"
        partnerLinkType="lns:Customer"
        myRole="cus4sell"/>
     <partnerLink name="CarrierIP"
        partnerLinkType="lns:Carrier"
        myRole="cus4carr"/>
</partnerLinks>

<variables>
        <variable name="PurchaseOrderID" messageType="lns:POMessage"/>
        <variable name="purchaseAcceptance"  messageType="lns:InvMessage"/>
        <variable name="ClockX"  messageType="lns:ClockX"/>
</variables>

<sequence>
     <invoke partnerLink="Seller"
        portType="lns:custosellPT"
        operation="sendPurchaseOrder"
        variable="PurchaseOrderID">
     </invoke>
     <request partnerLink="Seller"
        portType="lns:custosellPT"
```

```
        operation="ReceivePurchaseAcceptance"
        variable="purchaseAcceptance">
      </request>

      <if><condition> purchaseAcceptance == "true" </condition>
        <then>
            <sequence>
                <assign>
                    <copy>
                        <from> <literal> "00:00" </literal> </from>
                        <to variable=ClockX/>
                    </copy>
                </assign>
                <request partnerLink="Carrier"
                        portType="custosellPT"
                        operation="SendAcceptance"
                        variable="purchaseAcceptance">
                </request>
            </sequence>
         </then>
      </if>
      <if><condition> purchaseAcceptance == "false" </condition>
        <then>
            <empty>
            </empty>
        </then>
      </if>
</sequence>
</process>
```

B.2.3 The Seller Process

```
<process name="InternetPurchaseOrderSellerProcess"
        targetNamespace="http://acme.com/ws-bp/Internetpurchase"
        xmlns="http://schemas.xmlsoap.org/ws/2004/03/business-process/"
        xmlns:lns="http://manufacturing.org/wsdl/Internetpurchase">
```

```
<documentation xml:lang="EN">A simple example of a WS-BPEL process
for handling a Internet purchase order.</documentation>

<partnerLinks>
     <partnerLink name="CustomerIP"
        partnerLinkType="lns:Customer"
        myRole="sell4currIP"/>
     <partnerLink name="CarrierIP"
        partnerLinkType="lns:Carrier"
        myRole="sell4carr"/>
</partnerLinks>

<variables>
        <variable name="PurchaseOrderID" messageType="lns:POMessage"/>
        <variable name="purchaseacceptance"  messageType="lns:InvMessage"/>
        <variable name="ClockX"  messageType="lns:ClockX"/>
</variables>

<sequence>
     <receive partnerLink="Customer"
        portType="lns:custosellPT"
        operation="receivePurchaseOrder"
        variable="PurchaseOrderID">
     </receive>
     <if>
        <condition>
            ClockX  < "2:00"
        </condition>
        <then>
            <pick>
                <sequence>
                    <assign>
                        <copy>
                            <from>
                                <literal>
                                    "true"
```

```
                    </literal>
                </from>
                <to variable=purchaseAcceptance/>
            </copy>
        </assign>
        <invoke partnerLink="Customer"
            portType="custosellPT"
            operation="SendAcceptance"
            variable="purchaseAcceptance">
        </invoke>
    </sequence>
    <empty>
        <documentation>
            to implement a self loop equal to
            a pick with two activities an
            empty activity and the self loop
            activity
        </documentation>
    </empty>
</pick>
</pick>
    <if>
        <condition>
            purchaseaccepted == "true"
        </condition>
        <then>
            <invoke partnerLink="carrier"
                portType="selltocarrPT"
                operation="PickUpOrder"
                variable="PurchaseOrderID">
            </invoke>
        </then>
    </if>
    <if>
        <condition>
            purchaseaccepted == "false"
```

```
                    </condition>
                    <then>
                        <sequence>
                            <assign>
                                <copy>
                                    <from>
                                        <literal>
                                            "false"
                                        </literal>
                                    </from>
                                    <to variable=purchaseAcceptance/>
                                </copy>
                            </assign>
                            <invoke partnerLink="Customer"
                                portType="custosellPT"
                                operation="SendNoStock"
                                variable="purchaseAcceptance">
                            </invoke>
                        </sequence>
                    </then>
                </if>
            </pick>
        </then>
    </if>
</sequence>
</process>
```

B.2.4 The Carrier Process

```
<process name="InternetPurchaseOrderCarrierProcess"
        targetNamespace="http://acme.com/ws-bp/Internetpurchase"
        xmlns="http://schemas.xmlsoap.org/ws/2004/03/business-process/"
        xmlns:lns="http://manufacturing.org/wsdl/Internetpurchase">

<documentation xml:lang="EN">A simple example of a WS-BPEL process
for handling a Internet purchase order.</documentation>
```

```
<partnerLinks>
     <partnerLink name="SellerIP"
        partnerLinkType="lns:Customer"
        myRole="carr4sell"/>
     <partnerLink name="CustomerIP"
        partnerLinkType="lns:Carrier"
        myRole="carr4cus"/>
</partnerLinks>

<variables>
        <variable name="PurchaseOrderID" messageType="lns:POMessage"/>
        <variable name="ClockX"  messageType="lns:ClockX"/>
</variables>

<sequence>
     <receive partnerLink="Seller"
        portType="lns:selltocarrPT"
        operation="receivePickUp"
        variable="PurchaseOrderID">
     </receive>
     <if>
        <condition> ClockX  < "24:00" </condition>
        <then>
            <invoke partnerLink="Customer"
                portType="carrtocusPT"
                operation="DeliverPurchase"
                variable="PurchaseOrderID">
            </invoke>
         </then>
     </if>
</sequence>
</process>
```

Bibliography

[1] Luca Aceto, Augusto Burgueño, and Kim Guldstrand Larsen. Model checking via reachability testing for timed automata. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 263–280, London, UK, 1998. Springer-Verlag.

[2] R. Alur and D. Dill. Automata for modeling real–time systems. In *In Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, volume 443. Springer–Verlag, 1990.

[3] Scott W. Ambler. *The Object Primer 3rd Edition: Agile Model Driven Development with UML 2.* Cambridge University Press, 2004.

[4] J. Anderson, S. Fickas, and W. Robinson. The kate project: Supporting specification construction. In *Technical Report CIS-TR-90-24, Department of Computer and Information Scence, University of Oregon, Eugene, Oregon 97403*, 1990.

[5] Assaf Arkin, Sid Askary, Ben Bloch, and et. al. Web services business process execution language version 2.0, December 2004. http://www.oasis-open.org/committees/download.php/10347/wsbpel-specification-draft-120204.htm.

[6] Assaf Arkin and et al. Web service choreography interface (wsci) 1.0. In http:/www.w3.org/TR/wsci/.

[7] 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.

[8] Christel Baier. *On Algorithmic Verification Methods for Probabilistic Systems.* PhD thesis, Faculty for Mathematics and Informatics, University of Mannheim, 1998.

[9] F.L. Bauer and et al. The munich project cip, vol. 1: The wide spectrum language cip-l. *Lecture Notes in Computer Science 183*, 1985.

[10] G. Behrmann, T. Hune, and F. Vaandrager. Distributed timed model checking - How the search order matters. In *Proc. of 12th International Conference on Computer Aided Verification*, 2000.

[11] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, Yi Wang, and C. Weise. New Generation of UPPAAL. In *Int. Workshop on Software Tools for Technology Transfer*, June 1998.

[12] 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. In Rajeev Alur and Thomas A. Henzinger, editors, *CAV96*, number 1102 in LNCS, pages 244–256. Springer–Verlag, July 1996.

[13] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic and non-deterministic systems. Number 1026 in LNCS, 1995.

[14] T. Bolognesi and E. Brinksma. Introduction to the iso specification language lotos. *Computer Networks and ISDN Systems 14*, pages 25–59, 1987.

[15] J.P. Bowen and M.G. Hinchey. Ten commandments of formal methods ... ten years later. *IEEE Computer, 39(1)*, pages 40–48, January 2006.

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

[17] H. Bowman, G. Faconti, and M. Massink. Specification and verification of media constraints using UPPAAL. In *In 5th Eurographics Workshop on the Design, Specification and Verification of Interactive Systems*, number 1384 in Eurographics Series, pages 281–297. Springer–Verlag, August 1998.

[18] R.S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, 1979.

[19] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In *Proc. 1998 Computer-Aided Verification, CAV'98*, volume 427. LNCS.

[20] Marius Bozga, Oded Maler, and Stavros Tripakis. Efficient verification of timed automata using dense and discrete time semantics. In *Conference on Correct Hardware Design and Verification Methods*, pages 125–141, 1999.

[21] Mario Bravetti, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Supporting e-commerce system formalization with choreography languages. In ACM Press, editor, *In Proc. of the 20th ACM Symposium on Applied Computing (SAC'05)*, 2005.

[22] Mario Bravetti, Roberto Lucchi, Gianluigi Zavattaro, and Roberto Gorrieri. Web services for e-commerce: guaranteeing security access and quality of service. In ACM Press, editor, *In Proc. of the 19th ACM Symposium on Applied Computing (SAC'04)*, 2004.

[23] P. Broadfoot and G. Lowe. On distributed security transactions that use secure transport protocols, 2003.

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

[25] R.M. Burstall. An informal introduction to specification using clear. *The Correctness Problem in Computer Science*, pages 185–213, 1981.

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

[27] D. Cazorla, F. Cuartero, V. Valero, F. Pelayo, and J. Pardo. Algebraic theory of probabilistic and nondeterministic processes. *The Journal of Logic and Algebraic Programming*, 2002. To appear.

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

[29] K.M. Chandy and J. Misra. *Parallel Program Design.* Reading, MA: Addison-Wesley, 1988.

[30] M.H. Cheheyl, M. Gasser, G.A. Huff, and J.K. Miltolen. Verifying security. *ACM Computing Surveys 13, 3*, pages 279–340., September 1981.

[31] 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.

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

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

[34] Luc Clement, Andrew Hately, Claus von Riegen, and Tony Rogers. Uddi version 3.0.2, 2004. http://uddi.org/pubs/uddi_v3.htm.

[35] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer, 1985.

[36] T. Constable and et al. *Implementing Math. with the Nuprl Proof Developement Enviroment.* Prentice Hall, 1986.

[37] P. Cooper, D. Eckmann, J. Gingerich, S. Holtsberg, N. Kelem, and R. Martin. *FDM User Guide*. Unisys Corporation, Cul- Feiertag79 ver City, CA, 1889.

[38] R. Corin and S. Etalle. An improved constratint-based system for the verification of security protocols. *Lecture Notes in Computer Science*, 2477:326–341, 2002.

[39] R. J. Corin, S. Etalle, P. H. Hartel, and A. Mader. Timed analysis of security protocols. *Journal of Computer Security*, 2006. Imported from DIES.

[40] D. Craigen and et al. m-eves: A tool for verifying software. Unisys Corporation, Cul- Feiertag79 ver City, CA, 1889.

[41] A. Dardenne, A.van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.

[42] P. D'Argenio, B., H. Jensen, and K. Larsen. Reduction and refinement strategies for probabilistic analysis. In *Process Algebra and Probabilistic Methods - Performance Modelling and Verification, PAPM-PROBMIV 2002*, volume 2399 of *LNCS*, Copenhagen (Denmark), July 2002.

[43] P. D'Argenio, B. Jeannet, H. Jensen, and K. Larsen. Reachability analysis of probabilistic systems by successive refinements. In *PAPM-PROBM 2001*, volume 2165 of *LNCS*, Aachen (Germany).

[44] R. Darimont and A.van Lamsweerde. Formal refinement patterns for goaldriven requirements elaboration. In *Ninth International Workshop on Formal Methods for Industrial Critical Systems, (FSE-4 - 4th ACM Symp. on the Foundations of Software Engineering)*, October 1996.

[45] Luca de Alfaro. *Formal Verification of Probabilistic Systems*. Phd thesis, Stanford University, 1997.

[46] G. Delzano and S. Etalle. Proof theory, transformations, and logic programming for debugging security protocols. *Lecture Notes in Computer Science*, 2372:76–90, 2001.

[47] G. Diaz, D. Cazorla, F. Cuartero, and V. Valero. P_uppaal a tool for capturing the probabilistic behaviour of uppaal models. In *Journal: XI Jornadas de Concurrencia*, 2003.

[48] G. Diaz, D. Cazorla, F. Pelayo, F. Cuartero, and V. Valero. Verifying and capturing probabilistic bechaviours of real-time systems. In *19th Annual UK Performance Engineering Workshop*, 2003.

[49] G. Diaz, F. Cuartero, and K. Larsen. P_uppaal a tool for capturing the probabilistic behaviour of uppaal models. Technical Report DIAB–02–01–33, Computer Science Department, University of Castilla - La Mancha, 2002.

[50] G. Diaz, J. J. Pardo, M. E. Cambronero, V. Valero, and F. Cuartero. Automatic translation of ws-cdl choreographies to timed automata. *Lecture Notes in Computer Science*, 3236:230–242, 2005.

[51] G. Diaz, J. J. Pardo, M. E. Cambronero, V. Valero, and F. Cuartero. Verification of web services with timed autoamata. *Electronic Notes in Theoretical Computer Science*, pages 19–34, 2006.

[52] Gregorio Díaz, Fernando Cuartero, Valentín Valero Ruiz, and Fernando L. Pelayo. Automatic verification of the tls handshake protocol. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 789–794. ACM Press, 2004.

[53] Gregorio Díaz, Kim Guldstrand Larsen, Juan José Pardo, Fernando Cuartero, and Valentin Valero. An approach to handle real time and probabilistic behaviors in e-commerce: validating the set protocol. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 815–820. ACM Press, 2005.

[54] E. W Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[55] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[56] Bruce Eckel. *Thinking in Java.* Prentice Hall PTR, Upper Saddle River, 1998.

[57] H. Ehrig and et al. *An Algebraic Specification Language with Two Levels of Semantics.* Tech U Berlin 83-03, February 1983.

[58] Internet Engineering Task Force. The tls protocol version 1.1. *work in progress*, June 2003. http://www.ietf.org/internet-drafts/draft-ietf-tls-rfc2246-bis-05.txt.

[59] J.A. Goguen. Obj as a theorem prover with applications to hardware verification systems.

[60] M. Gordon. Hol: A proof-generating system of higer-orderlogic. In *VLSI Specification, Verification and Synthesis.* Kluwer, 1987.

[61] Jim Grundy. Report on m-EVES. ERL Research Report ERL-0545-RR, Information Technology Division, Electronics & Surveilance Research Laboratory, Defence Science & Technology Organisation, Building 171 Laboratories Area, PO Box 1500, Salisbury SA 5108, Australia, March 1991.

[62] Guttag and Horning. *Larch: Languages and Tools for Formal Specification.* Springer, 1993.

[63] J.V. Guttag, J.J. Horning, and J.M. Wing. Some remarks on putting formal specifications to productive use. *Science of Computer Programming, Vol. 2, NO. 1*, pages 53–68, October 1982.

[64] Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and et. al. Soap version 1.2 part 1: Messaging framework, 2003. http://www.w3.org/TR/soap12-part1.

[65] Nicolas Halbwachs. Delay analysis in synchronous programs. Number 697 in Lecture Notes in Computer Science. Springer-Verlag, 1993.

[66] D. Harel. On visual formalisms. *Comm. ACM 31 5*, pages 514–531, May 1988.

[67] Klaus Havelund, Kim Guldstrand Larsen, and Arne Skou. Formal verification of a power controller using the real-time model checker UPPAAL. *Lecture Notes in Computer Science*, 1601:277–298, 1999.

[68] 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. In *Proceedings of the 18th IEEE Real–Time Systems Symposium*, pages 2–13. IEEE Computer Society, 3–5 December 1997.

[69] I. Hayes. *Specification Case Studies.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

[70] Constance Heitmeyer and Dino Mandrioli. *Formal Methods for Real-Time Computing.* John Wiley & Sons, 1996.

[71] M. Hennessy. *Algebraic Theory of Processes.* Cambridge, MA: MIT Press, 1988.

[72] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. What good are digital clocks? In *Automata, Languages and Programming*, pages 545–558, 1992.

[73] A. Heydon and et al. Miro: Visual specification of security. *IEEE Transactions on Software Engineering, vol. 16(10)*, pages 1185–1197, 1990.

[74] C.A.R. Hoare. *Notes on Data Structuring.* Academic Press, 1972.

[75] C.A.R. Hoare. *Proof of Correctness of Data Representations.* Acta Informatica, Vol. 1, No. 1, 1972.

[76] C.A.R. Hoare. *Communicating Sequenctial Processes.* Prentice Hall Int'l, 1985.

[77] H. Hüttel, K. G. Larsen, and C. Weise. The use of static sontructs in a modal process logic. Number 363 in Lecture Notes in Computer Science. Springer-Verlag, 1989.

[78] IEEE. Ieee standard vhdl language reference manual, 1988.

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

[80] H. Jensen, K. Larsen, and A. Skou. Modelling and analysis of a collision avoidance protocol using spin and uppaal, 1996.

[81] B. Jonsson and K. Larsen. Specification and refinement of probabilistic processes. Amsterdam, Holland, July 1991.

[82] B. Jonsson, K. Larsen, and W. Yi. Probabilistic process algebra. E-HPA, chapter 4: Extensions, pages 685–710. 2001.

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

[84] 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).

[85] Nickolas Kavantzas and et al. Web service choreography description language (wscdl) 1.0. http://www.w3.org/TR/ws-cdl-10/.

[86] R.A. Kemmerer and S.T. Eckman. A user's manual for the unisex system. In *Tech. Report Dept. of Computer Science, Univ. Calif.*, December 1983.

[87] R. Koymans, J. Vytopil, and W.P. de Roever. Real-time programming and asynchronous message passing. In *Proc. Second ACM Symp. Principles Distributed Programming*, pages 2187–197, 1983.

[88] L. Lamport. The 'hoare logic' of concurrent programs. *Acta Informatica 14*, pages 21–37, 1980.

[89] A.van Lamsweerde. Requirements engineering in the year 00: a research perspective. In *International Conference on Software Engineering*, page 5–19, 2000.

[90] C. Landwehr. *The Best Available Technologies for Computer Security.* Computer 16, 7, July 1983.

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

[92] K. Larsen, P. Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[93] K. Larsen, P. Pettersson, and Wang Yi. Uppaal in a nutshell. *Journal on Software Tools for Technology Transfer*, 1, 1997.

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

[95] Kim G. Larsen, Paul Pettersson, and Wang Yi. Diagnostic Model–Checking for Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in LNCS, pages 575–586. Springer–Verlag, October 1995.

[96] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL: Status and developments. In Orna Grumberg, editor, *CAV97*, number 1254 in LNCS, pages 456–459. Springer–Verlag, Jun 1997.

[97] F. Larsson, K. Larsen, P. Pettersson, and Wang Yi. Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction. In *Proc. of the 18*th *IEEE Real-Time Systems Symposium*, 1997.

[98] P. Lee and et al. The ergo support system: An integrated set of tools for prototyping integrated inveroments. In *Proc. of the Third ACM SIGSoft Symp. Software Development Enviroments*, pages 25–34, November 1985.

[99] P. Lescanne. Computer experiments with the reve term rewriting system generator. In *Proceedings Tenth ACM Symposium on Principles of Programming Languages*, pages 99–108, 1983.

[100] K.N. Levitt, L. Robinson, and B.A. Silverberg. *The HDM handbook*. Thech. Report Vols. 1-3, SRI Int'l, 1979.

[101] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gear-Box Controller. In *Proc. of the 4*th *Workshop on Tools*

and Algorithms for the Construction and Analysis of Systems, number 1384 in Lecture Notes in Computer Science, pages 281–297. Springer–Verlag, March 1998.

[102] J.L. Lions. Ariane 5: Flight 501 failure, July 1996. In http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf.

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

[104] G. Lowe. casper: A compiler for the analysis of security protocols. In *10th IEEE Computer Security Foundations Workshop (CSFW'97)*, pages 18–30. IEEE Computer Society, 1995.

[105] G. Lowe. Towards a completeness result for model checking of security protocols. In *Proc. of The 11th Computer Security Foundations Workshop*, 1999.

[106] David Luckham and et al. *ANNA - A Language for Annotating Ada Programs.* Springer, 1987.

[107] M. Matsuda, M. Sato, and Y. Ishikawa. Efficient implementation of portable c*-like data-parallel library in c, 1997.

[108] P.R. McMullin and J.D. Gannon. Combining testing with formal specifications: A case study. In *IEEE Trans. Sfotware Eng., Vol. 9, No.3*, May 1983.

[109] A.J.R.G. Milner. A calculus of communicating systems. *Lectures Notes in Computer Science 92*, 1980.

[110] R. Nakajima and et al. *The Iota Programming System.* Springer, 1983.

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

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

[113] D.L. Parnas. A technique for the specification of software modules. *Comm. ACM 15, 5*, pages 330–336, May 1972.

[114] J.L. Peterson. Petri nets. *Computing Surveys, Vol. 9, No. 3*, September 1977.

[115] A. Pnueli. The temporal logic of programs. In *Eighteenth Annual Symposium on the Foundations Sysof Computer Science. Long Beach, CA: IEEE Computer Society*, November 1977.

[116] A. Pnueli. Applications of temporal logic to the specifications and verification of reactive systems: A survey of current trends. *Lecture Notes in Computer Science*, 224:510–584, 1986.

[117] C. Ponsard, P. Massonet, A. Rifaut, J.F. Molderez, A. van Lamsweerde, and H. Tran Van. Early verification and validation of mission critical systems. In *Ninth International Workshop on Formal Methods for Industrial Critical Systems*, 2004.

[118] P.R. D'Argenio, J.P. Katoen, T.C. Ruys, and J. Tretmans. The bounded retransmission protocol must be on time! In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 416–432, Enschede, The Netherlands, 1997. Springer Verlag, LNCS 1217.

[119] M.L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Willey series in probability and mathematical statistics. Jhon Wiley and Sons, 1994.

[120] R. Alur and T.A. Henzinger. Logics and Models of Real-Time: A Survey. In *Real Time: Theory in Practice*, volume 600, pages 74–106. Springer-Verlag, 1991.

[121] C. Rich, T.C. Waters, and H.B. Reubenstein. Toward a requriments apprentice. In *Proc. Fourth Int'l Workshop Software Specification and Design*, pages 79–86, april 1987.

[122] L. Robinson and O. Roubine. Special – a specification and assertion language. *Tech. Report CSL-46, Stanford Research Inst., Menlo Park, California*, Jannuary 1977.

[123] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

[124] D.S. Rosenblum and D.C. Luckham. Testing the correctness of tasking supervisors with tsl specifications. In *Porc. ACM SIGSoft 89 (Third Symp. Software Testing, Analysis and Verification)*, pages 187–196. No. TAV-3, 1989.

[125] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *Modelling and Analysis of Security Protocols.* Addison Wesley, 2001.

[126] D. Sannella and A. Tarlecki. Program specification and development in standard ml. In *Proc. Symp. Principles Programming Languages*, pages 66–77, 1985.

[127] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems.* Phd thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, June 1995.

[128] James F. Stay. HIPO and integrated program design. *j-IBM-SYS-J*, 15(2):143–154, 1976.

[129] W. Richard Stevens. *The Protocols TCP/IP Illustrated, Volume 1.* December 1993.

[130] W. Swartout. The gist behavior explainer. In *Proc. Am. Assoc. Artifial Intelligence Conf.*, pages 402–407, august 1983.

[131] D. E. Thomas and P. R. Moorby. *The Verilog Hardware Description Language, Third ed.* MA: Kluwer Academic Publishers, 1996.

[132] D.H. Thompson and R.W. Erickson. *AFFIRM Ref- programerence Manual.* USC Information Sciences Institute, Marina del Rey, CA, February 1981.

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

[134] Mastercard & VISA. Set secure electronic transaction specification: Business description, 1997.

[135] Mastercard & VISA. Set secure electronic transaction specification: Formal protocol definition, 1997.

[136] Mastercard & VISA. Set secure electronic transaction specification: Programmer's guide, 1997. http://www.setco.org/set specifications.html.

[137] Sanjiva Weerawarana, Roberto Chinnici, Martin Gudgin, and et. al. Web services description language (wsdl) version 2.0 part 1: Core language, 2004. http://www.w3.org/TR/2004/WD-wsdl20.

[138] J.M. Wing. A specifier's introduction to formal methods. *IEEE Computer, 23(9)*, pages 8–24, September 1990.

[139] Eurostat yearbook 2004. *The statistical guide to Europe. Data 1992-2002.* European Commission: EUROSTAT, Office for Official Publications of the European Communities, 2004.

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

[141] P. Zave. *PAISLey User Documentation, Volume 1: REFERENCE MANUAL.* AT&T Bell Laboratories, 1987.