Temporal Logic Models for Distributed Systems

by

Cristian Lambiri

A thesis submitted to the
School of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Masters of Applied Science

Ottawa-Carleton Institute for Electrical Engineering
Department of Electrical Engineering
Faculty of Engineering
University of Ottawa
August 1995

ISBN 0-612-04946-9

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

I hereby declare that I am the sole author of this thesis.

I authorize the University of Ottawa to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Cristian Lambiri

I further authorize the University of Ottawa to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Cristian Lambiri

ABSTRACT

Since the beginning of the 1980's, the way the computer systems are conceived has changed dramatically. This is a direct result of the appearance, on a large scale, of personal computers and engineering workstations. As a result, networks of independent systems have appeared.

This thesis presents a formal specification framework that can be used in the design of distributed systems. The abstract models that are presented are based on a systemic view of distributed systems and discrete event systems. Two base abstract models called *deterministic discrete event systems (DDES)* and *discrete event automaton (DEA)* are presented. For the *DEA* the series and parallel compositions as well as feedback connection are defined. Universal algebra is employed to study the parallel composition of DEAs. From the DDES/DEA an abstract model for distributed systems is obtained. Subsequently, linear time temporal logic is modified for use with the abstract chosen model of distributed systems. The logic is described in three aspects: syntax, semantics and axiomatics. The syntax is modified by the addition of two operators. The semantics of the logic is given over the abstract models. Five axioms are added to the axiomatic system for the two new operators.

A programming language called *TLL*, based on the theoretical framework, links the theory with practice. The syntax and semantics of the programming language are presented. Finally an example of modeling in the framework is given.

ACKNOWLEDGMENT

I wish to express my sincere gratitude to my supervisor, Dr. Dan Ionescu, for his constant guidance, encouragement, and support throughout my research.

I would like to thank my family and especially my wife, Eugenia Mihaela, for her love and caring.

Contents

List of Tables

List of Figures

Notation

Various symbols, superscripts, subscripts, and abbreviations used frequently in this thesis are summarized below. All notation is fully defined where it first arises in the text.

Symbols

E	Set of events.
T	Set of time values.
t	The time variable.
Q	Set of states.
Y	Set of output values.
I	Input alphabet.
q	State of a DDES or DEA.
(e, t)	Timed event.

Greek Letters

δ	Next state function.
β	Output function.
Γ	Set of logical formulas.
ε	The null event.
ϕ	The emply set.
μ	Processing time.
ς	Trajectory.

Special Symbols

$\oplus$	Parallel Composition of DEAs.
$\odot$	Serial Composition of DEAs.
$\neg$	The logical negation symbol.
$\wedge$	The disjunction connective.
$\vee$	The conjunction connective.
$\rightarrow$	The conditional implication.
$\leftrightarrow$	The biconditional implication.
$\bigcirc$	Logical next.
$\Box$	Henceforth operator
$\Diamond$	Eventually operator
$\mathcal{U}$	The until operator.
$\|\|$	Parallel operator.
$\mid$	Sequential operator.

Acronyms and Definitions

iff	If and only if
ADT	Abstract data type.
BNF	Backus-Naur Form.
DES	Discrete Event System.
DDES	Deterministic Discrete Event System.
DEA	Discrete Event Automata.

LALR(1)	Look Ahead 1 token, left to right input processing , right most derivation grammar.
OS	Operating System.
POSET	Partially Ordered Set.
TLL	Temporal Logic Language.
TS	Telephone System.
$\mathcal{DS}$,DS	Distributed System.
P	Set of Processes.
C	Set of communication channels.

Chapter 1

Introduction

1.1 Motivation and Research Objective

With the advent of low cost, high speed computer networks, distributed systems are an ever more important part of the computing environment. Most of these systems are extremely complicated and therefore adequate specification methods are important to make their design process more manageable. Several advantages make the formal method attractive [42]:

- They can detect ambiguities, omissions and contradictions that can occur in the informal specifications.
- A formally verified part can be embedded in a system with more confidence, thus reducing the testing time.
- A formal model can be the basis for an automated design tool.
- Several designs can be compared.

The above advantages are well known to the computer community. The number of available theories and corresponding practical frameworks attest this. Most of them do not deal specifically with distributed systems, but they can be adapted for such use.

Formal frameworks can be used in several ways. The language provided by the framework can be used to specify a system and this seems to be the most utilized approach. Several frameworks also come with axiomatic systems, which provide a way to verify a specification. The drawback of such an approach is that the verification can be at best only partially automated for non

trivial problems. The designer then directs the theorem prover in search of a solution. Thus, the designer has to know not only the syntax and semantic of the specification language, but also an entire axiomatic system, to be able to direct the theorem prover towards the desired goal.

What is lacking from all the frameworks taken into consideration is a theory built on a systemic model of distributed systems. Such a theory can employ systemic notions to define several *general* properties that are desirable in distributed systems. By doing this, the class of *valid* models for a specification is restricted. Thus, verification of the system is performed at two levels: firstly over its model, by imposing restrictions on the class of acceptable models and secondly over its logical description, where we can make a verification of correctness with the help of a deductive system.

The purpose of this thesis is to develop a framework for the specification of distributed systems. Several levels of abstraction can be taken into consideration for such a framework. The first one deals with the *concrete model* which we choose for the system under study. Another level of abstraction consists of an *abstract model.* This model is based, in our case, on a systemic view of distributed systems. This point of view allows us to impose restrictions on the systems that we consider as having a desired behavior.

The thesis also investigates a logical framework, based on temporal logic. Temporal logic was chosen due to several motivations:

- It is well developed and continues to be an active area of research.
- Several variants,that can be used for our purposes, are available, with different syntaxes and semantics.
- Temporal logic has already been applied to specification, verification and synthesis of concurrent programs thus presenting a large base of research in the domain.

1.2 Organization of the Thesis and Contributions

The rest of the thesis is organized as follows: Chapter 2 presents an overview of the existing literature on the subject of formal specification. Due to the richness of the subject, a presentation of all the frameworks that are currently in use or under investigation is not possible. Thus, only those relevant to our position are presented.

In Chapter 3, the base for our framework is presented. Thus, this chapter deals with the subject of models for distributed systems. A systemic abstract model,that forms the base for all the future work, is built in the chapter. For the abstract model, several system compositions are defined. Thus, from the base model, more complex abstract models are built, using composition. The result is the definition of our abstract model for distributed systems.

Chapter 4 introduces a variant of temporal logic that represents the base of our specification language. The language has its syntax modified by the addition of two new operators. The semantics is defined over the abstract model of distributed systems, introduced in Chapter 3. The proof system has been modified accordingly to accommodate the new operators.

The theoretical framework developed in the previous chapters is put to work in a programming language. Chapter 5 presents the syntax of our programming language, called *TLL* , along with its semantic. The semantic of the language is based on the logical language presented in Chapter 4.

Chapter 6 gives an example of the use of the programming language. The example consists of modeling a telephone system and specifying the model in *TLL* . The thesis ends with Chapter 7, which contains conclusions and future work directions.

This thesis contains several research contributions, which can be summarized as follows:

1. A new theoretical framework for modeling distributed systems, that applies a systemic point of view, is built in Chapter 3. Three types of compositions over the base abstract model are presented.

2. In Chapter 4, classical temporal logic is modified by adding two new operators that seem better suited to represent the true parallelism of distributed systems. Also, the semantic of the logic is given over the abstract model of distributed systems developed in Chapter 3.

3. The syntax of *TLL* , the programming language presented in Chapter 5. The last contribution is related to the semantic description of *TLL* that is given using the theoretical framework developed in the Chapters 3 and 4.

Chapter 2

Theories and Models

The purpose of this chapter is to present some theories and models for concurrent and distributed systems that appear in the literature. The literature on the subject is vast and thus, only works considered relevant to the subject are presented. The chapter starts with an explanation of the meaning of *distributed systems* in Section 2.1. Sections 2.2 and 2.3 present the current state of the domain of formal specifications for concurrent and distributed systems. An important aspect of any abstract theory is the *model* of that theory. Several concrete and abstract models are presented in Section 2.4. The chapter ends with a small compendium of definitions from universal algebra, that will be used in the following chapters. Section 2.5 might look misplaced, but universal algebra is a powerful tool in the study of formal models and thus, the section was added to this chapter.

2.1 Distributed Systems

The term *distributed system (DS)* has different meanings in the computing community, depending on the specialization of the person that uses it [47]. As the word *distributed* implies, a *DS* refers to a system made up of components which do not reside at the same place. This might seem confusing because *same place* can have different connotations depending on the scale that one uses to make his/her observations. From the software community's point of view, the following definition best describes a distributed system.

Definition 2.1 *A **distributed system** is a spatially spread system that, through the set of processes that are run by the system, gives the impression to the user that the system is a unit.* ■

The notion of a unit is related to the fact that the user should **not** see where a resource is located or on which particular machine a program is executed.

Several characteristics, that surface from Definition 2.1 [77], can be summarized as follows:

- A DS is composed of multiple independent computers. An *"independent"* computer is a computing machine that can execute computations independently and therefore has its own processor(s), storage space and input-output capabilities. Their physical characteristics confer to these systems the status of truly parallel systems. Therefore parts of the system work independently and simultaneously. This characteristic is used mainly to make a distinction between parallel and distributed systems.

- The components of a DS are interconnected. Without interaction, a set of components cannot be called a system. In the computer world, interaction is realized through communication. Therefore, we need to have a set of communication channels between the parts of the system. The border with parallel systems is blurry in this area. Usually, distributed systems are considered to be those that use a communication subsystem, other than a bus, for information exchange between the parts of the system.

- The components of a DS cooperate towards a common goal. Communication does not imply cooperation. What sets a distributed system apart from other communicating systems is that we perceive it as a unit. Therefore, a user cannot make any distinction between various machines that are part of the system. Cooperation also imposes a need for coordinating and synchronizing the various subparts (otherwise it would not be a system). Thus, the designer has to provide means of synchronization between the parts. The need for communication is satisfied in several ways, but the paradigm of message passing seems the most natural due to the nature of the communication channels.

The fact that we have several interconnected computers does not mean that we necessarily have a distributed system. The third point in the above description is crucial because it makes the transition from simply networked systems to distributed systems. In a way, *distributed computing systems* try to make a link between two different worlds: the centralized systems and the networked systems. The latter have the advantage that they are easily scalable and allow sharing of spread resources. Nevertheless, the repartition is not hidden from the user, who has to know where the resources are located in order to use them. The centralized systems

are more attractive because they are easier to use. They give the user the impression that all the resources are readily available. The same thing may not be true for networked systems. Presently, the communication channels are the bottlenecks in any networked system. Therefore, they impose more problems than the rest of the system. Major problems are generated by the three characteristics of today's communication systems [77]: unreliability, insecurity and high cost. The machines that compose the network might have problems too, in the form of independent failure. Often, only one machine crashes, while the others still work. If we do not consider independent failure, the whole system might be brought to a stop. System reliability is worsened in a networked system that does not allow independent failure by the introduction of more failure points.

2.2 Specification and implementation

The design process, as any human activity, has to be partitioned in manageable sub tasks, in order to be tractable. Hommel [34] presents the following steps, which have to be followed by a designer, to produce a working system starting from a problem statement:

- Specify *what* the software should do. This step is the statement of the problem as a whole.
- Partition the problem into subproblems. To make a problem tractable, we usually apply a decomposition process that translates one big problem into several smaller problems. Once we have the partitions we can specify *what* the modules that will solve the subproblems should do and *what* is the logical relation between them.
- Specify *how* each module should work. This step is the actual problem solving step, when we tackle all the problems and try to come up with solutions (algorithms) to them.
- Write the whole system in a machine translatable programming language. Of course, the purpose of any system built upon a general programmable computer is a computer executable program. Therefore, we have to translate the algorithms from the specification language into a machine executable language.
- Verify that the implementation satisfies the specification. If the specification is automatically translated into an implementation, this last step is no longer needed. Usually, this step is called validation.

The first three steps in our enumeration are usually called the *specification* phase while the last one is called the *implementation* phase. It has to be said, though, that the use of the word *specification* varies, although slightly, with different authors. Most of them use it to describe all the sentences that define a system, as others [37] use it to describe only those sentences that eliminate unwanted behavior from a specification. It might be argued that since the elimination of the unwanted behavior is still part of the system description, it can be considered that both variants have essentially the same meaning. Thus, the *specification* of a system is the set of statements that describe how the system should behave. Specifications should be restrictive enough that they present one system behavior and only one, but also general enough that they are not biased towards a certain implementation. To achieve this, it is necessary [41] to specify at least the interface between the system and the environment (the interface that makes the data exchange with other entities of the system).

During the implementation phase, a specification is translated into a workable format, which can be either a program or a hardware design. In software design, the transition from specification to implementation is just a translation from one language (the specification language) to another (the programming language). The ideal thing would be to automate this translation and therefore to use only a single language (the specification language) throughout the development process.

Validation is the last phase of the design process. This phase ensures that the implementation and the specification have the same semantic meaning. This can be done either by case testing or by formal proof. Case testing is tedious and time consuming because test cases have to be written. Formal proofs can be used to show that the implementation is correct. Formal verification has its roots in a paper written by Floyd in 1967 [18] where he defines a method of proving *partial correctness.*

The specification phase is crucial because the final product is always compared against the specification. Any mismatch between the specification and the implementation of a piece of software is called a *bug* and testing can usually detect it. If the specification is incorrect, we have a *design error* that can be more difficult to detect before the product is delivered since the test cases are also generated from the specification.

Specifications can be formal (i.e. based on mathematical language) or informal (i.e. based on a natural language). In both cases, the semantics of the specifications have to be unambiguous. However, they also have to have enough expressive power to allow us to describe the properties of

the system. Natural languages are the most expressive languages yet, because of their ambiguous nature, specifications using them are prone to inconsistencies. As an example, one can see that the specification for Novell's Service Advertising Protocol (SAP) [68] has contradictory statements in it. Although more prone to error, the informal specification has the undeniable advantage of being easy to learn and use.

To fix the ambiguity problem that the informal specifications exhibit, the research community turned its attention to formal specification languages. These are, in essence, languages derived from mathematics, which have been adapted for the computer science community. Their mathematical background make them difficult to learn and use for the average designer. On the other hand, they give the user the advantage of mathematical object manipulation and thus, they allow the designer to prove the correctness of *specifications.*

Related with the notion of specification language is that of logical reasoning. Logic allows humans to manipulate notions of various objects. Coupling a specification language with an axiomatic system gives us the ability to produce a powerful reasoning mechanism. However, doing this solves only half of the problem. We also need to have an interpretation of the language A, which we use, in the universe of the problem that we try to solve. Usually this is done by giving an interpretation of the language A in another language B, which already has defined a semantics in the world of the problem.

2.3 Theories describing concurrent and distributed systems

Concurrency is an important aspect of computing. Thus, it comes at no surprise that its theoretical study generated a great deal of interest. The term *concurrent system* denotes a system that executes several processes at the same time, or gives such an impression to the user. The semantic varies slightly as the word *parallel* is also used, usually for systems that present true temporal simultaneity. By the more lax semantics, which includes the parallel system in the class of concurrent systems, distributed systems are a special class of concurrent systems. Hence, the models and theories that exist for concurrent systems are important for the study of distributed systems.

The theories that have emerged in the field of concurrent systems can be classified in several groups based on different criteria. First, we partition them based on the way they handle time.

Some do not consider time and are therefore, *untimed* theories. Some consider it and are called *timed* theories.

An early example of an *untimed* theory is *Petri's net theory*. Developed by C. A. Petri in the mid 1960's [66] this theory is based on a particular kind of directed graph called a *Petri net*. The graph has two kinds of nodes called *places* and *transitions*. The arcs in the graph are from a transition to a place or vice versa. Each *place*, graphically represented by a circle, is marked with a nonnegative integer t, represented graphically by dots, signifying the number of *units of information* for that node. *Places* are linked with *transitions*, represented as bars. The arcs are also marked with nonnegative numbers signifying the number of parallel links that the arc represents. Formally, we can define Petri's structure in the following way:

Definition 2.2 *A Petri net is a 5-tuple* $PN = (P, T, F, W, M_0)$, *where:*

- P *is a finite set of places.*
- T *is a finite set of transitions.*
- $F \subseteq (P \times T) \cup (T \times P)$ *is a set of arcs,*
- $W : F \longrightarrow \mathsf{N}$ *is a weight function.*
- $M_0 : P \longrightarrow \mathsf{N}$ *is the initial marking.*
- $P \cap T = \phi$ *and* $P \cup T \neq \phi$

■

The notion of *firing*, which is the process of generating tokens in new places, characterizes the dynamic behavior of Petri nets. A transition occurs when certain preconditions, regarding the number of tokens that have to be contained in the *places* that are connected to that *transition*, are fulfilled. Some tokens, which will be placed in the next *place*, are generated by a postcondition after the *firing* occurs.

Petri nets can be used to model different systems, including finite state machines, communication protocols, formal languages and parallel activities. Depending on the way in which transitions are connected with places Petri nets can model either deterministic or nondeterministic processes. The appeal of the method is the fact that it uses graphical representations that

are easy to understand. Petri net theorists classify the properties that can be studied with this formalism as *behavioral* and *structural* properties.

The behavioral properties, such as *reachability, observability, liveness, fairness and safety*, are fundamental to the study of any type of system and are independent of the method of investigation. *Reachability* allows one to decide if parts of the system are useless, in the sense that they cannot be reached during normal functioning of the system. The *safety* property asserts that something bad does not happen during the functioning of the system, while the *liveness* property assures that something good will eventually happen. *Fairness* is closely related with the notion of concurrency. It assures that if two or more processes are running concurrently on the same processor each of them will eventually run if it is enabled.

Although Petri net theory has many followers due to its qualities, it also has its critics. The most important criticism brought up [34] is that although it can represent flow control or data flow, Petri net theory cannot handle both simultaneously.

Another group of *untimed* theories is composed of *algebraic process theories.* They are called algebraic because they are built upon a set of rules (constructors, combinators) by expressing equivalences through *equations.* The most important members of this family are the following:

- Hoare's *Communicating Sequential Processes or CSP* [33]. *CSP* is more closely related to some programming languages with parallel constructs and unbuffered synchronous communication. Programs have the form $[Pl_1 :: C_1 \parallel \ldots \parallel Pl_n :: C_n]$ where $Pl_1 \ldots Pl_n$ are labels for the processes $C_1 \ldots C_n$. Process communication is realized through two primitives: send ($P!E$, process P sends event E) and receive ($P?A$ process P receives A). Both primitives have to be executed simultaneously.

- *Calculus of Communicating Systems (CCS)* developed by Milner [62, 63] and *Algebraic Theory of Processes (ATP)* which was developed by Hennessey [29]. Both these theories, are built upon the same model [63](*Labeled Transition Systems*) and are based on behavioral study of processes. The differences are in the way the authors define the observation process. CCS studies the equivalence through the notion of *bisimilarity* as opposed to the ATP that defines the notion of *testing.* Hennessy [30] showed that the two notions are related. Both theories consider processes equivalent if, they have the same set of actions and if both processes enter in a deadlock after the execution of an action.

C.A.R. Hoare is credited with the first attempt to use logic as a formal system in computer science consistently. Since Hoare's fundamental paper [32] on the axiomatization of computer programs, an enormous quantity of work has gone in this direction. In his paper, Hoare defined correctness formulas for sequential programs as: $\{P\}C\{Q\}$ where C is the program and P, Q are called *precondition* and *postcondition*, such that if P is satisfied at the beginning of the program Q will be satisfied at the end of the program. Hoare logics are *endogenous* logics, meaning that its programs are explicit in the language.

The logic that is of most interest for this thesis is *temporal logic*. The term denotes a *class* of logics that can be used to reason about time. TL is a successor of *tense logic*, a branch of mathematical logic, whose origins we can trace to Arthur Prior. His logic system [75] tried to deal with the tense modalities that appear in languages.

The language of temporal logic, like its relative, modal logic, uses modalities like *sometime* and *always* to deal with the *possibility* of truth in the future. The subject started to be of interest for the computer science community in the 1970's and has received considerable interest ever since.

As with any language, temporal logic variants are constructed by specifying a syntax and a semantics. The syntax of the language gives the rules of construction for the valid propositions of the language. The semantics is used to give a definition of the truth in the logic theory, usually over a suitable mathematical structure. Semantic of the *tense logic* was given over *frames* [83]:$(T, <)$ where T is a set of points of *time* equipped with a binary precedence relation $<$. The models for tense logic use these time frames to generate a structure (usually called a *Kripke structure*): $(T, <, V)$ where V is a valuation assigning each proposition p a set of time points where it holds. Modified variants of the above structure are used to define the semantics of temporal logic too. Depending on the interpretation of time by the language, temporal logic semantics can be classified as:

- *Interval semantics.* In this interpretation the time is composed of intervals (periods). This view allows the *chopping* and the reunion of periods [65, 67]. Allen [4] has argued that interval semantics is the most efficient way to carry temporal information in temporal data bases.

- *Point semantics.* It views time as consisting of points. The set that defines the time can

be either *sparse* or *dense*. This view is more common in computer science because can make the relation with the notion of state, where *past* and *future* are described as relations between states.

The two views are related by the fact that a period of time can be viewed as a collection of points, while a point can be viewed as a limit of a period. Point semantics can be further sub divided into:

- *Linear time* semantics. Under this choice a set of detached runs of a program P are a *set of sequences* of the states of the program. This was the first current in temporal logic whose main proponents are Manna and Pnueli [54, 53, 56, 55]. Concurrency of the processes is represented by the interleaving of the atomic steps of the program. For example, if we have two atomic steps a and b in two different processes A and B, the following two executions are equivalent: $a \parallel b = ab + ba$, where xy denotes the sequential execution of x and y, the "+" sign signifies the logical *or* and $\parallel$ denotes the concurrent execution of the steps. The underlying structure of time has the following properties [21]:

  - it is discrete,
  - it has an initial moment, which does not have any predecessors and
  - it is infinite into the future.

  This structure is similar with that of the natural numbers set as defined by Peano's set of axioms. Thus, $\mathbb{N}$ is usually used to represent the time in this semantic. The semantics of linear time temporal logic is given over a generalized Kripke structure. Here, states replace worlds [21, 80] and sequences of states replace the binary relation on worlds. The sequence of states is also called a *full path*, *computation sequence* or simply *computation*. The syntax of the language is built from [56]: atomic propositions, boolean connectives and temporal operators.

- *Branching time* semantics [16],[22],[23]. In this case a tree of alternatives represents the runs of the system. There are several sub variants of this kind of logic, the most important being CTL^* by Emerson and Clarke [17]. If the *linear time temporal logic* has interpretations only on states, the *branching time* version has modalities on paths too. If we have a concurrent program made of several processes the number of ways in which the program can continue

with the execution from a certain point is nondeterministic. Therefore if we consider every alternative as happening in a different world we can view the time as "splitting" into several branches. Each node can have infinitely many successors. The structure of time is a tree with every path in the tree corresponding with a time line. Each time line is isomorphic with N. Therefore, time is considered discrete and sparse. While *linear time temporal logic* considers every execution sequence individually, the *branching time* version considers them all at once in a tree like structure.

The syntax differs between different variants. $CTL^{\star}$ syntax, which is the most expressive variant, accepts two types of formulas: *state* formulas and *path* formulas formed from propositions, boolean connectives, temporal operators and existential quantifiers.

- *Partial order* semantics. In 1984 [73], Pinter and Wolper introduced a new variant of temporal logic that used partial order semantics. The original authors [73] and others, have argued that this semantics is suitable to reasoning about distributed computations.[39, 41]. The syntax of their language [73] is a variant of CTL, except that it allows two backward path operators: previous and since. The semantics is given over the same time structure as for the *branching time* logic except that the accessibility relation acts in both directions.

Although it has a strong case in the domain of specification, TL is not without its faults. First, temporal logic, at least in its original form, does not allow quantitative reasoning about time. Several researchers have tried to cope with this deficiency and modify the language such that it allows the specification of time stamps in the expressions [37, 57]. Another critic directed against it was that it does not allow composition. Barringer [8] solved this problem by proposing a compositional system that allows modular reasoning about programs: one can verify a program by verifying its constituents and afterwards combine the parts.

Computing systems, due to their digital nature, can be viewed as *discrete event systems.* A *discrete event system* is a dynamical system whose evolution in time is characterized by the followings:

- Events occur instantaneously, and at discrete times only.
- Systems are event-driven.
- Usually, operation of the system is non-deterministic.

The discrete time nature of DES made them suitable for study using a point based TL [70, 37].

Since its introduction to computer science temporal logic has been used in several ways [21]. The first usage was that of manual program composition followed by manual program verification. Another usage involved the axiomatization of temporal logic and the proof of the interesting properties as axioms in the proof system. These two uses can be viewed as verification of concurrent programs. Another mainstream approach was the synthesis of programs from specification. This type of approach is interesting because if the specifications are correct we don't have to validate the generated program anymore.

- *Specification* language. Temporal logic has been used as a specification language for concurrent systems [55, 70]. This usage is usually combined with another activity and therefore is not considered as a stand-alone application, but it is nevertheless important.

- *Verification of concurrent programs.* This was the original usage of temporal logic as advocated by Pnueli in his original paper [74]. Since then, this application has seen a diversification of methods used for it. All of the variants of temporal logic were used as languages [49] [56]. Several techniques were used:

  - automatic verification
  - axiomatic verification

- *Program synthesis* from temporal logic specifications [12] [13],[11],[16] [52] [15]. Program synthesis tries to produce programs in a machine translatable language from a non executable specification. Usually this activity is reduced to the synthesis of finite state programs because of the decidability problem. Because the generated finite state machines are considered *models* of the specification, this activity is also called *model checking.* This use also includes a number of executable programming languages based on temporal logic as are [69]: *tempura, tokio, templog and chronolog*

From a theoretical standpoint, the two approaches to temporal logic are related: one tries to develop an axiomatic system for a certain model and the other tries to find a model that suites a certain theory.

2.4 Concrete and abstract models

The modeling of a system is the abstraction process. The result of this process is an object that best reflects the properties of the modeled system. Once obtained, the model allows designers to predict the behavior of a system before they build it.

A *model* [76] is a collection of attributes and a set of rules that show how these attributes interact. When mathematical objects compose the set of attributes, the model is called *abstract*. *Concrete* models are those that have as attributes characteristics of other existing systems.

To build a satisfactory model of a distributed system, we have to know what properties this type of systems exhibit. Communication between components is crucial in making the system to act as a unit and, therefore, the way we perceive the information exchange is an important step in the modeling process. Based on that, Lamport [47] classifies concrete models as:

Message Passing models. One can imagine this model, probably the most natural candidate for distributed systems, as a black box that has a set of input lines and a set of output lines. The communication between two processes is done by *sending* and *receiving* messages. Several issues that have to be dealt with to have a working model in this paradigm:

- *Network topology.* A distributed system can be viewed as a collection of processes that communicate with each other. The processes can be abstracted as the nodes of a communication graph where the arcs are the communication channels. Therefore, the system can be viewed as a graph, directed or undirected, depending on the interpretation of the communication channel. However, an undirected graph can be represented by a directed graph with more arcs and therefore the type of graph used in the model is not important, at least from a theoretical point of view.

- *Synchrony.* Depending on how much synchronism is needed between the sender and receiver we can impose several restrictions on the model. At one extreme of the spectrum is the *asynchronous model* that does not consider time. By adding time, which can be done in several ways varying from a timer to partial relation of events, the model can be modified to have a certain level of synchrony. The *asynchronous model* makes no assumption regarding the timing of the process execution or the message passing delays. Because the physical characteristics, spatial distribution without a common clock, makes the system asynchronous, adopting this model means imposing no restraints at all.

- *Failure.* The nature of the system incurs several points of failure, but we have two classes of failure points: *process failures* and *communication channel failures.* Models can be classified based on the types of failures that they consider [76]. The weakest condition that can be imposed is that the failure of a process does not affect the other processes and therefore, the rest of the system can detect the failure. Imposing a greater uncertainty we can eliminate the detection condition from the requirements. Another class of failure models considers the *omission* of the messages that have been sent to them. Finally, the most restrictive models are the *Byzantine failures* models. In this set-up, the failed processes exhibit an undefined behavior. Lamport et al. [48] proved that this problem has a solution in certain cases.

- *Message buffering.* Between the time a message is sent and the moment it is received there is a delay attributed to the communication channel and to the message buffering. On a high speed network, though, the network delay due to transmission is small compared to the delay due to buffering. Depending on the cardinality of messages that can be stored in the buffer we can consider either a *finite* or an *infinite* buffering system. If the number of buffers is considered finite the system may exhibit message losses due to the buffer full condition. Obviously any real system has only a finite buffer capacity, but if the dimension of the buffer is much greater than the transmission capacity we can consider the buffer infinite. The buffer's discipline is also important for the behavior of the system. The simplest case is the FIFO when the messages are processed in strict arrival order, but we can also consider models that accept out-of-band data that are closer to the transport level protocols.

Shared variables models. In this model, the processes communicate through a common memory zone. This model was in broad usage for classical concurrent systems and has the advantage of being well understood, with applications that are relatively easy to program. This model also allows the use of well known and understood synchronization methods such as semaphores and monitors. However, for distributed systems, there has to be a layer of software that makes the abstraction to this model from message passing and this can incur a performance penalty.

Synchronous communications models. Hoare [33] introduced this type of communication in his work about *communicating sequential processes (CSP).* This way of communication resembles message passing but as opposed to it, the send and receive primitives are totally synchronous

(i.e. the owning processes execute them in the same time). "Normal" message passing systems do not require this condition.

Another aspect of a distributed system model regards the structure of the process space. A fundamental model for the distributed systems (and not only them) is that of the *client/server* model. The motivation for this model is based on our perception of how multiple independent entities should work. In this setting the *client* is the part of the system that makes requests for services. The *server* is an entity that waits for a request to be made, processes it and gives back an answer. Servers can be classified in several ways. One possible classification divides them into *concurrent*, if they spawn another process for each client that connects to them and *sequential* if the messages are processed in a loop. We also have to consider the quantity of information that a server keeps during a connection. So called *state* servers are the ones that maintain information about connections with clients. *Stateless* servers do not maintain such information. State servers have potential problems in case of crashes because they cannot easily recover to the precrash state. In this view, the system is made of a client and a server. Lin [37] proved that having only the specifications for the system and the client, one can determine by projection the specification for the server.

An interesting variant of synchronous communication is the *remote procedure call* (RPC). The rationale behind the system is to simplify the software paradigm by hiding the communication between processes through the use of procedure calls. This is done by using a *client/server* system. The procedure call that some process makes is processed and transformed into a message that is subsequently sent to the server for execution. Hence, the programmer is spared of all the problems generated by the programming of the communication subsystem. Thus, distributed programs are placed in the same paradigm as the classical concurrent programs (i.e. those that run on a centralized system).

Formal specification, as its name implies, uses the language of mathematics to specify a precise meaning for programs. To use formal specification techniques, once we have decided on the concrete model, we need to find an abstract model which can describe the behavior of the concrete model. Several abstract models in use, depending on the theory built upon them: *labeled transitions systems* [62, 63, 29, 80], *extended state machines* [70, 71], *fair transition systems* [56],*basic transition systems* [55], *temporal logic models* [50, 37], *temporal structures* [21] and *live I/O automata* [79].

Labeled transition systems are derived from Kripke structures, by specifying the worlds as states and the binary relation as a transition relation. The set of labels ensures the link with the theory whose model the transition system is chosen to be.

Definition 2.3 *A **labeled transition system** (LTS)* $(S, T, \{\xrightarrow{a}: t \in T\})$ *where: S is a set of **states**, T is a set of **transition labels** and* $\xrightarrow{a} \subseteq S \times S$ *is a* transition relation *for each* $t \in T$. ■

From the above definition, Emerson [21] derived its *temporal structures.* Depending on the structure of time that they convey, there are two variants: *linear time temporal logic* and *branching time temporal logic.* The difference between them is the way in which they define the succession of states.

Mana and Pnueli [55] use models that are based on the notion of variable as a memory location in order to provide a semantics for their temporal logics.

Definition 2.4 *A **basic transition system** is a quadruple* $(\Pi, \Sigma, \tau, \phi)$ *where: Π is a set of state variables, Σ is a set of states, τ is a finite set of transitions, and ϕ is an initial condition.* ■

The *fair transition system* [56] is a more restrictive definition of the *basic transition system* including also two families of *requirements* called: *weak fairness requirements* and *strong fairness requirements.* Ostroff [70] defined *extended state machines* which are a variant of *basic transition systems* that include the communication primitives.

Other models have been tried. The most interesting are those of Andersen [79] and Lin [37]. Andersen et al. use a special type of automata [79] as a model for their temporal logic. Lin, in his PhD dissertation [37] applied universal algebra to the study of his abstract model. This is a pattern that will also be followed in this thesis. The advantage is that a link can be readily made between a formal model and a specification language.

2.5 Universal Algebras

Many researchers have noted the importance of universal algebra for computer science. Büchi [10] argued that many notions from automata theory were just alternative expressions of the same thing: algebraic properties. In the mid 1970's, Goguen realized the importance of universal algebra to abstract data type specification. Since then, a vast literature on the subject has arisen.

In our case, the universal algebra is important because it allows us to link a proposed model of distributed systems with a logical framework that can describe them.

The connection can be made through a branch of mathematics called *model theory* that gives a relationship between the syntax of a logical system and its semantics (model). The relationship was concisely summed up by Chang and Kesler in [14]: **universal algebra + logic = model theory**. More recently, Goguen and Burstall [27] presented a generalization of model theory by introducing the notion of an *institution*, an abstraction of the notion of a *logical system*. This section introduces basic notions from universal algebra that will be used in the development of the model in Chapter 3. Let us start our small presentation by defining two notions that will be used throughout this section: *arity* and *signature*.

Definition 2.5 *An **n-ary operation** on the set A is a function $f : A^n \longrightarrow A$, $f(a_1, a_2, \ldots, a_n) = a$, where $a, a_1, \ldots, a_n \in A$. The number n is called the **arity** of f.* ■

Definition 2.6 *A **signature** is a set Σ whose elements are called **operation symbols**, together with an arity function $ar : \Sigma \longrightarrow \mathsf{N}$.* ■

Depending on the number of sets over which it is defined, a universal algebra can be *single sorted* or *multi sorted* [20, 84]. In the literature, both are called Σ-*algebras* because they are defined over signatures. The notion of **sort** is not explicitly defined anywhere in the literature known to myself, but it seems to define a collection of symbols that are used to index sets.

Definition 2.7 *Let Σ be a signature. A **Σ-algebra** is a pair $\mathbf{A} = (A, \Sigma_A)$, where A is a set called the **carrier** of A , and $\Sigma_A = f_A | f_A : A^n \longrightarrow A, n \in \mathsf{N}, ar(f) = n$ a set of functions.* ■

There are several examples of *single sorted algebras*: groups, monoids and rings are all algebras. An important point about the above definition is that the functions that compose the algebra have to be total. A field, for example, is *not* a universal algebra. In order to make this concept useful we need to introduce the notion of structure-preserving mapping, or homomorphism, between **single sorted algebras.**

Definition 2.8 *Let $\mathbf{A} = (A, \Sigma_A)$ and $\mathbf{B} = (B, \Sigma_B)$ be two **single sorted algebras**. A homomorphism of A into B is a function $\phi : \mathbf{A} \longrightarrow \mathbf{B}$ such that, $\forall f \in \Sigma$ we have:*

$$\phi(f_A(a_1, \ldots, a_k)) = f_B(h(a_1), \ldots, h(a_k)), \forall a_1, \ldots, a_k \in A$$

■

Central to the concept of algebra are the notions of expression and equation. Expressions can be build by syntactically adding subexpressions using the symbols allowed by the signature of the algebra. Therefore, we can generate a set of *words* (expressions) over signatures and variable sets. The set of all expressions generated by a set of variables X over a signature Σ, denoted $T_\Sigma(X)$ can be made into an algebra.

Definition 2.9 *Let Σ be a signature and X be a set of variables. The algebra $Term_\Sigma$ with carrier $T_\Sigma(X)$ and operations defined by: $\sigma(t_1, \ldots t_n) = \sigma t_1 \ldots t_n$ for any n-ary operation symbol $\sigma \in \Sigma$, is called a eterm algebra over X.* ■

The induction principle used to prove theorems over the height of terms is called *Principle of term induction* and is applied as follows: to prove that a property P holds for all terms in $T_\Sigma(X)$, it suffices to show the validity of the following properties:

- P holds for all variables and constants.
- If P holds for any terms $t_1, \ldots, t_n$ in $T_\Sigma(X)$, then P holds for $\sigma t_1 \ldots t_n$ for all $\sigma \in \Sigma$ and $n \geq 1$

Definition 2.7 can be generalized, in the form used in computer science by Goguen [61], to the many sorted case. Let S be a set of sorts. Let $A = \{A_s | s \; in S\}$ be an S-sorted set. The set A is said to be finite if the disjoint union over all its components is finite and A_s is nonempty for only finitely many indices $s \in S$. An S-sorted signature is a set Σ, whose elements are called operation symbols, together with an arity function $ar : \Sigma \longrightarrow S^* \times S$, such that $ar(\sigma) = (w, s)$, where $\sigma \in \Sigma$ and w is a word over S called the arity (domain) of σ and $s \in S$ is called the range of σ.

Definition 2.10 *A multi sorted algebra Σ is a pair (A, Σ_A) where A is a **family** of S-sorted sets and Σ_A is a **family** of operations such that each operation symbol $\sigma : w \longrightarrow s$ of Σ is realized as an operation $\sigma^A : A^w \longrightarrow A_s$.* ■

The structure preserving mappings for the multi sorted case are also called homomorphisms. For the multi sorted case the homomorphism is defined as follows:

Definition 2.11 *Let* **A** *and* **B** *be two multi sorted algebras. A mapping from* $h : \mathbf{A} \longrightarrow \mathbf{B}$ *is a* Σ*-homomorphism if* $h_s(\sigma^A(x)) = \sigma^B(h_w(x))$ *for all operation symbols* $\sigma : w \longrightarrow s$ *in* Σ *and all* $x \in A^w$. ■

The relationship between *temporal logic* and *universal algebra* was first made explicit by Lin [37]. It should be noted though that the author confuses multi sorted and single sorted versions of universal algebras. As a starting point for his work he cites Hennessy's [29] definition of universal algebra. The problem is that the definition, which Lin also uses, is for single sorted algebras and therefore, all function realizations have to be totally defined over the carrier set of the algebra which is not true for his model. Technically, all his results from Chapter 6 [37] would be correct for the multi sorted version of universal algebra as defined by Goguen [61]. This is due to the fact that the realizations are defined over *components* of the carrier of the algebra, as is the case for his *temporal logic models.*

Chapter 3

Models of Real-Time Distributed Systems

The first step in building a framework suitable for specification is to build a model for the object of study. As mentioned in the previous chapter, two kinds of models will be considered: concrete and abstract models. The chapter starts with considerations about the concrete model in Section 3.1. The theoretical base of discussion is given in Section 3.2 by the introduction of the dynamic system. A model for time and events is subsequently presented in Section 3.3. All these three elements, *system, time* and *events* are essential in a model for distributed systems, as it will be seen in this chapter. In Sections (3.4, 3.5 and 3.6) several abstract models are built and discussed. Finally, in Section 3.7 an abstract model for distributed systems is presented. This will be the basis for a temporal logic approach to system description in subsequent chapters.

3.1 Concrete Model

Concrete models are an important part of any modeling activity, because they can help to form a view about the object that at study. The term *concrete* is used to differentiate the model from the *abstract* model, which is as previously said just a mathematical structure.

There are two fundamental views regarding a system that have to be considered: they can be either **closed** or **open.** A system is considered **closed** if it does not have any *external* interactions. This view, advocated mainly by Lamport [45, 1, 46], but also by others [79], starts from the assumption that are no external interactions but that there are *internal* ones between

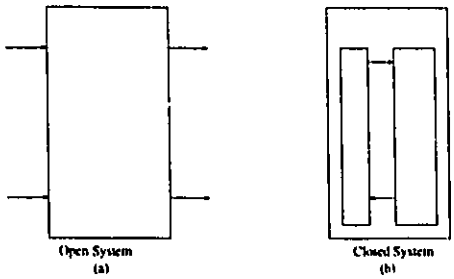


Figure 3.1: Open and Closed Systems

parts of the system. Evidently, this is a global approach to specification that considers the whole system. Nevertheless, it presents the advantage of simplifying the specification through a "action-reaction" principle": every output from a part of the system is an input to another (Figure 3.1b).

The other point of view is to consider the system **open** to *external* interactions. In this case every system exerts actions on other systems through its outputs and accepts external signals through its inputs. A more complex system can be obtained from simpler parts by composing them, using several operations. This point of view has the advantage that allows specifications to be reused, by modularization. In the sequel, the systems will be considered **open** and thus, subject to external interaction (Figure 3.1a).

The concrete model to be considered below derives from the view of distributed systems as collections of processes linked together by communication subsystems. Such a system can be considered a double $(\mathcal{P},\mathcal{C})$, where $\mathcal{P}$ denotes the set of processes and $\mathcal{C}$ the set o communication systems. Processes are running concurrently, possibly on several machines, and respond to external events either by actions on the environment or by data exchange with other parts of the system. Evidently, the system is open because it interacts with the environment through events.

There are two fundamental ways to realize data exchange between processes: through message passing or through shared memory. Message passing is the "native" mode of behavior for any system that does not have a common address space for its parts. Although well understood and simpler to program, the shared memory model cannot be directly implemented on spatially distributed systems. The designer has to rely on other methods to deliver the illusion of a common address space. This is usually done by considering another layer of abstraction above a message

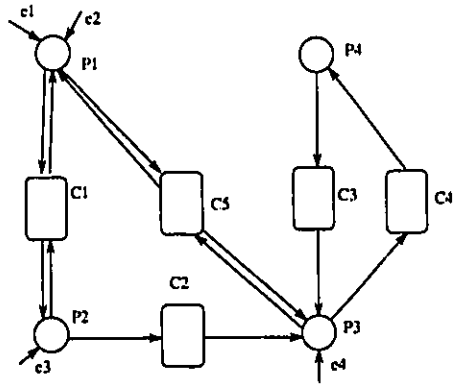


Figure 3.2: Distributed system

passing system.

Considering the processes and the communication subsystems as nodes of a graph, a distributed system can be abstracted as a bipartite graph with two types of nodes (Figure 3.2). One type of node denotes the processes that compose the system and the other the communication subsystem. To exchange information, two processes have to be connected through a communication system that is responsible for the delivery of messages to the destination.

To reduce the design complexity of the communication systems, several layering stacks for the communication protocols have been proposed. The most common is the OSI layering architecture that separates a communication system in seven layers. Usually, what is studied from the point of view of the software systems are the logical characteristics of the protocols that realize and maintain the connection at various levels in the stack. To make the problem of protocol design more tractable, all communication protocols will be considered to have only a finite number of states and to be event driven.

Therefore, all the specification details for the communication systems can be moved in separate modules. This implies that from the point of view of the processes only the reliability of the communication system counts. Our concrete model will consider only reliable communication systems, which always deliver messages. Although this seems like a strong assumption, it is valid at least in the local area networks, where the message corruption is small.

If real-time criteria are important in a distributed system, the interaction with the environment has to be responsive [40]. This means that the response time to external stimuli has to be

bounded and the maximum processing delay known. To achieve this, two main approaches have been used in the design of real-time distributed systems [40]:

- **Event triggered** approach. Such a system waits for something to happen, and when it does it takes the appropriate actions. The system will remain in the same state until an event appears at its inputs.
- **Time triggered** approach. This system reacts to events only at predefined moments in time. Events, though, can appear anytime. However, they are buffered and processed only when the system is enabled for work.

Asynchronous systems are, of course, *event triggered* and any synchronous system that is clock driven is *time triggered.* Hybrids of the model are also possible, for example by considering *event triggered* systems that buffer inputs for processing. Our model considers only processes that are event driven and have discrete states. These characteristics make them suitable for formalization as *discrete event system.* Communication systems are also event driven (in this case the events are the messages) and have discrete states. So, they can also be studied as *discrete event systems.*

3.2 Dynamical Systems and Automata Theory

The search for an abstract model starts from the notion of *dynamical system* that is the most general notion of system known to me. Together with classical automata theory, this will form the background for the abstract model.

Automata theory has several faces, depending on the angle that one wants to see it [6]. The most common trends, in the study of automata theory, are to regard automata as:

- **Constructs.** This aspect is concerned with the relation between parts of an automaton (analysis) and the way several automata can be linked together to perform a task.
- **Languages.** It is the most studied aspect of automata theory, probably because of the practical implications to the implementations of translators. The main concern here is with finding automata that recognize sets of symbols called *languages.*
- **Dynamical systems.** Arbib [38] made the link between two fields that, up to that moment, where considered different: automata theory and system theory. Systemic properties such

as reachability, observability and controllability can be studied about of automata and so general conditions on the correctness of an automaton can be imposed.

- **Algebraic systems.** The mathematical structure of automata made them ideal candidates for study with the help of algebra. The study evolved from the use of group theory to the more modern view involving universal algebra.

Each of these aspects can be investigated separately or in combination, depending on the use of such a theory. The prevailing aspect in the study of automata theory was and continues to be that of automata as *language acceptors*[10, 28, 35]. Even in the field of specification of concurrent and real-time systems, where general properties of the system should be more important, this seems to be the case [31, 5]. Brandin and Wonham [9] made an interesting combination by studying automata as *dynamical languages* and *language acceptors.*

Distributed systems are made up of several components that work together. In this case, a view of automata as *constructs* can be helpful in building more complicated structures. By considering them *dynamical systems* one may impose conditions that define some forms of correct behavior. The algebraic view can help in making the link with logics through the notion of model.

The formalization effort starts from the systemic notion of *dynamic system*, as defined by Kalman [38]. A system in his view is any structure into which something is put at certain times and from which something is output at certain times. The *state* of the system is an internal attribute, which together with the input, determines the subsequent behavior of the system.

Definition 3.1 *A* dynamical system *[38] is a mathematical structure:* $S = (T, X, Y, \Omega, Q, \Gamma, \varphi, \beta)$ *where*

- T *is a set of* time *values,* $T \in (\mathsf{R}, \leq)$, Q *a set of states,* X *is a set of input values,* Y *is a set of output values,* $\Omega = \{\omega : T \longrightarrow X\}$ *a set of admissible input functions and* $\Gamma = \{\gamma : T \longrightarrow Y\}$ *is a set of output functions that satisfy the following conditions:*
  - Ω *is nonempty.*
  - *If* $\omega_{[t_0,t)} \in \Omega$ *and* $[t_1, t_2) \subset [t_0, t)$ *then* $\omega_{[t_1,t_2)} \in \Omega$.
  - *If* $t_1 < t_2 < t_3$ *and* $\omega, \omega' \in \Omega$, *then there is an* $\omega'' \in \Omega$ *such that* $\omega''_{[t_1,t_2)} = \omega_{[t_1,t_2)}$ *and* $\omega''_{[t_2,t_3)} = \omega'_{[t_2,t_3)}$.

- $\varphi : T \times T \times Q \times \Omega \longrightarrow Q$ *is a relation that satisfies* $q(t) = \varphi(t; \tau, q, \omega) \in Q$, *resulting at time* $t \in T$ *from* ***initial state*** $q = q(\tau) \in Q$ *at* ***initial time*** $\tau \in T$, *under the action of the input* $\omega \in \Omega$. φ *has the following properties:*

  - $\forall t_0 \in T, q \in Q, \omega \in \Omega : \varphi(t_0; t_0, q, \omega_{[t_0,t_0)}) = q$.
  - $\forall t_0, t_1, t$ *such that* $t_0 < t_1 \leq t$,

$$\varphi(t; t_0, q, \omega_{[t_0,t)}) = \varphi(t; t_1, \varphi(t_1; t_0, q, \omega_{[t_0,t_1)}), \omega_{[t_1,t)})$$

  - *If* $\omega, \omega' \in \Omega$ *and* $\omega_{(\tau,t]} = \omega'_{(\tau,t]}$ *then*

$$\varphi(t; \tau, q, \omega) = \varphi(t; \tau, q, \omega')$$

- $\beta : T \times Q \longrightarrow Y$, *is a* ***readout map*** *with* $y(t) = \beta(t, q(t))$. *The map* $[t_0, t_1) \longrightarrow Y$ *given by* $\beta(t, \varphi(t; t_0, q, \omega_{[t,t_0)}), t \in [t_0, t_1)$ *is an output segment, which is the restriction* $\gamma_{[t_0,t)}$ *of some* $\gamma \in \Gamma$.

■

The most important set in the above definition is the time set T. Time has been formalized in several ways. Classical automata theory considers it to be N [6], the set of natural numbers. Over this set Ω is specified by X and δ and β need only be represented over $Q \times X$. Time does not have to appear explicitly in the definition of the system. This abstraction expresses extremely well the behavior of untimed systems. For real-time systems, where time counts, this is not so. Time has to be present in the definition of the system explicitly.

The Definition 3.1 describes a general non-anticipatory system. This means that the system is causal: past evolution influences the future, but the reverse is not true. By taking φ to be a relation the system becomes state nondeterministic. In this type of system the next state is probabilistically determined from a set of possible states. **DES** are, usually, nondeterministic. This is useful for modeling existing systems, but we are interested in the design of new systems and therefore we want that our processes to be deterministic. The class of specifications, of *implementable* systems, includes in this case, only deterministic systems. Deterministic systems are a particular case of Definition 3.1 where φ is a single valued function.

The systems described by Definition 3.1 are also time variant. For such a system the states and events are time indexed, because the evolution of the system depends on the initial state from which the system started. The initial state, as it is generally the case, does not have to be the same for successive runs of the same system. Furthermore, events are also time indexed. Two events of the same type are not the same if they appear at different moments in time. This is due to the fact that, at different moments in time the same input can have different outcomes on the system, although the system is in the same state then the event appears. If the system gives the same response to a given input segment, when in a given state, independently of the time at which the experiment takes place the system is called *time invariant.* More precisely:

Definition 3.2 *A system S is* ***time invariant*** *if and only if* $\forall \tau, t_0 \leq t_1, \forall a \in Q$ *and* $\omega, \omega_1 \in \Omega$, $\omega_1(t) = \omega(t+\tau)$ *for* $t_0 \leq t \leq t_1$ *implies* $\varphi(t_1; t_0, q, \omega_1[t_0, t_1)) = \varphi(t_1+\tau; t_0+\tau, q, \omega[t_0+\tau, t_1+\tau))$ *and* β *does not depend on* t. ■

Automata, as defined in [6, 28, 35, 10] are time invariant systems. This is due to the nature of the time that is isomorphic with N.

3.3 Time and Events

The notion of system that we use in this thesis is time variant. Hence, we have to investigate the nature of time and events and to find suitable models for both this entities.

Time

Since its beginnings as a reasoning entity man has wondered about the nature of time. The understanding of time changed as knowledge evolved, culminating with Einstein's theory of relativity. The relation between different reference systems might or might not be linear, depending on their relative speed. We will consider our systems to be immobile or mobile at low speeds. In such systems the time is Newtonian, that is continuous, independent of our senses, elapsing in only one direction and in a linear relation between referentials. The most appropriate abstraction is the set of real numbers.

Digital systems, as it is known, have discrete states. The transition intervals between states, although not always equal, have an upper bound that can be taken as the transition interval

for the system. Technically, our connection with time is through a measuring system, because the smallest time value that we can discern is the value that we can measure. Let us choose this value as the unit of time. This leads to a set of time values that is isomorphic [6] with the set of natural numbers. Such a formalization consider the transition interval, the period of time needed to change state, to be one. This models monolithic systems well. Nevertheless, it creates problems when one tries to compose systems. Let us consider the composition of two such systems. The time for both systems is modeled by the set of natural numbers. However, although the speeds of the systems might be different, in general, we have to consider them to be the same, otherwise we might not be able to compose the systems. Considering the clock periods to be an arbitrary natural value n, instead of 1, might solve this problem. However, it is well known that the set of natural numbers is not closed under division and thus, we might not find a valid (in N) clock constant for the composed system. A time description based on *rational numbers* has to be adopted.

Ionescu [36] argued that, by taking a fine enough time constant, any system can be described this way. However, this introduces another problem. If parts of a specification are to be reused or modified, it can lead to the modification of time constants. In such a case, the whole specification has to be modified in order to make it consistent with the new value.

Let us choose the set of positive real numbers R^+ as our model for the set T and let us denote the transition interval of the system by μ, where $\mu \in \mathsf{R}^+ - \{0\}$. Over R^+ the usual binary operation of addition $(+)$ is also considered. The operation is associative in the usual way and thus $(\mathsf{R}^+, +)$ is a semigroup. By considering μ, the set of time values where the system can make transitions is isomorphic with N. If the system is also considered as *time driven* then the time intervals between two transitions are multiples of μ. This is not true for *event triggered* systems when the processing of an event can start anytime. Our model considers *event triggered* systems. Therefore, the number of steps that the system makes is *countable*. Here transition moments at least μ time apart. We call a set M *countable* if it is not finite and if a surjective map $s : \mathsf{N} \longrightarrow M$ from the set of natural numbers N onto M exists. We call the set *at most countable* if it is finite or countable.

Time is relative to the system of reference in which it is measured and so each system is considered to have its own *local time (LT)*. In order to be able to synchronize various parts of a distributed system a global entity called *global time (GT)* has also to be considered. Between any

LT and GT a Newtonian time relation exists. If τ is the GT moment when a system S started its functioning, considering a time moment in the two reference systems, t_g in GT and t_l in LT the relation between the two is: $t_g = t_l + \tau$.

Let us consider two parts P_1 and P_2 of the same system S. Each part has its own local time T_1 and T_2 respectively. Considering that both parts started to function at the same GT moment τ, there is obviously an isomorphism between T_1 and T_2: $id : T_1 \longrightarrow T_2$. This isomorphism implies that both systems measure the time in the same manner. Of course this is difficult to realize in a system, but fortunately there are several algorithms [82] that ensure the isomorphism condition.

Discrete event systems [37] have the property that events appear at discrete moments in time, but these moments can be anywhere on the time axis. R^+ is continuous from zero to ∞ and can, therefore, handle events that appear at any time moments.

Time moments are not the only kind of temporal structures. Of equal importance are the time intervals. We will denote a time interval $[t_1, t_2)$ by $\sigma_{t_1}^{t_2}$. Depending on the values of t_1 and t_2 several special cases that are more important will have separate denotations. A time interval that has only an upper limit $[0, t)$ will be denoted by σ^t. Similarly for a time interval that has only a lower limit, $[t, \infty)$, the following notation will be used σ_t. Single point intervals of the form $[t, t]$ will be denoted by $\bar{\sigma}_t$.

Let us consider now the set Σ of all time intervals over R. Over this set, let us define a partial operation , (denoted by +), called *interval concatenation*, in the following manner:

$$\sigma_{t_1}^{t_2} + \sigma_{t_2}^{t_3} = \sigma_{t_1}^{t_3}$$

,

$$\sigma_{t_1}^{t_2} + \sigma_{t_4}^{t_3} = \begin{cases} \sigma_{min(t_1,t_3)}^{max(t_4,t_2)} & \text{, if } [t_1, t_2) \cap [t_3, t_4) \neq \phi \\ \text{undefined} & \text{, if } [t_1, t_2) \cap [t_3, t_4) = \phi \end{cases}$$

Another way of introducing the time is by considering a global event called *clock* [70, 71, 9]. From a modeling perspective this is true because the clock system can be considered another subsystem that generates events to the main system. From a modeling perspective this looks natural but also complicates matters when defining the transition map, because special rules [9] have to be introduced in order to ensure the acceptance of the clock event. Other authors [46] advocate the use of time as just another variable that is part of the system. In this thesis the second view will be considered.

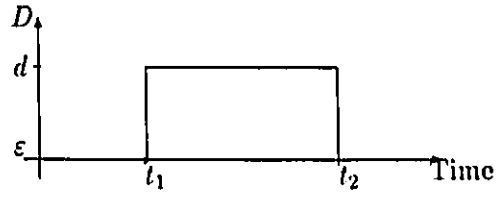


Figure 3.3: Timed events

Events

Once the structure of time has been established, it is the turn to investigate the structure of the events. Events can be classified in several ways. Following [36] we will partition the events into two broad categories: **timed** and **untimed**. Untimed events are considered asynchronous and do not have duration in time, thus resembling a Dirac function. The set of untimed events will be considered to be the set of symbols D. The symbols in this set can be considered as nullary functions $d : \phi \longrightarrow D$, where $d \in D$. The null event ε is considered to be part of D.

The set of timed events is a set of functions over T, the time set, with the codomain in D. This type of events are defined over an interval $[t_1, t_2)$ by the Equation 3.1. The lower time limit of the event can be interpreted as a time delay as the upper value can be interpreted as a deadline for the event to appear. As with their untimed counterparts the timed events are asynchronous and do not have time duration. However, the interval in which they appear regulates their effect on the system.

$$d_{\sigma}^{t_2}{}_{t_1}(t) = \begin{cases} d \in D - \varepsilon & \text{, if } t_1 \leq t < t_2 \\ \varepsilon & \text{, otherwise} \end{cases} \tag{3.1}$$

A particular case of timed events can model the untimed events. This is done by considering $t_1 = 0$ and $t_2 = \infty$. Considering this, the set of possible events is $E = D \cup E_T$. Timed events can be visually represented as depicted in Figure 3.3.

Events are described in their own reference system. To be more precise, if an event $e_{t_1}^{t_2}$ appears at moment t_d in the local time of a system, it has to be executed in the interval $[t_d + t_1, t_d + t_2)$ in order effect the system. The same interval in GT is given by $[\tau + t_d + t_1, \tau + t_d + t_2)$. Of course, an even more sophisticated interpretation can be brought up, where the event has a "good" behavior in the enabling interval and a "bad" one outside it. If the time interval is $[0, \infty)$ then

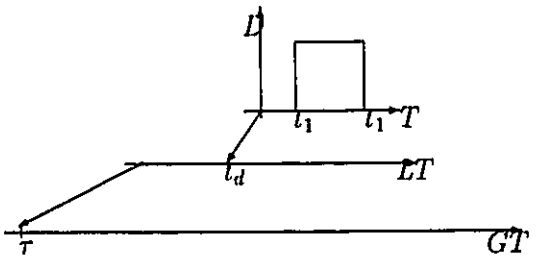


Figure 3.4: Relation between time referentials

the events are untimed. The interval addition can be used to simplify the event description. Let $e_d^{t_2}_{t_1}$ and $e_d^{t_4}_{t_3}$ be two timed events for the same symbol d. If the operation $\sigma_{t_1}^{t_2} + \sigma_{t_2}^{t_3}$ is defined, then $e_{d\,min(t_1,t_3)}^{max(t_4,t_2)}$ (Figure 3.4) represents both events.

The set of admissible input functions is defined over subsets of T and with the codomain in E. Events on the other hand are formalized as discussed above. The set of all the events that appear on any *finite* time interval has to have finite cardinality. This is due because our system reacts in a finite amount of time to any event. Thus, in the case that an *infinite* number of events are allowed to appear over a finite interval the system will not finish processing those events over a finite interval. Such a behavior is considered unacceptable and hence, the above condition is imposed. If $\omega_{t_1}^{t_0}$ is an acceptable input function defined over a time interval, with n the number of time points where ω is defined , the system will process all the inputs in a $n * \mu$ period of time. Ω is composed of functions ω defined in the following manner: $\Omega = \{\omega : T \longrightarrow E \mid \omega = e_k(t_{e_k})\}$. Thus, any function ω has to be defined in a *countable* set of time points. If $T_d \in T$ is the set of time points where the function ω is defined there exist a morphism $f : T_d \longrightarrow N$. If $Card(T_d) = \infty$ then f is an isomorphism.

The above condition ensures that the so called Zeno executions [79], over a finite interval, do not appear. In such a behavior, the processing time of a set of events that appear over a finite interval takes an infinite amount of time. There is also another type of Zeno behavior. This occurs when the number of events that appear over any finite interval is greater that the number of entries in the input buffer plus the number of events that can be processed in that interval. If K is the maximum number of events that can be buffered, the number of events that can appear over an interval $\sigma_{t_1}^{t_2}$ is given by: $N \leq K + \mid (t_2 - t_1)/\mu \mid$. If the number of events that appear over a period of time is greater than $K + [(t_2 - t_1)/\mu]$ then some of them will be lost. Over an

infinite interval this means that the arrival rate of the events has to be less than or equal to $1/\mu$.

The acceptable input functions can be viewed as words in a language L_E. The language is defined over an *infinite* alphabet $E \times T$ where E is a set of events (timed and untimed) and T is the time set. A word in the language w is defined as follows: $\forall \omega \in \Omega, \forall t \in T$ where $w = \omega(t_0)\omega(t_1)...\omega_{t_n}$. The language L_E is then composed of the set of all words w such that $w = e_0(t_0)e_1(t_1)...e_n(t_n)$ and $t_0 < t_1 < ... < t_n$, where $e_0, e_1, ..., e_n \in E$ and $t_0, t_1, ..., t_n \in T$. Hence, L_E is a subset of $(E \times T)^*$.

Another way of introducing timed events is by relating a *timer* with each event[9, 5]. In both [9] and [5] the authors present similar approaches in the way they introduce the timed events, with the difference that in [9] timers for events are integrated in the notion of state of the system as in [5] they are considered a separate entity.

3.4 Systemic Models of Discrete Event Systems

There are several formal models of *discrete event systems*. In this section we will consider a formal model for a *deterministic discrete event system (DDES)* by considering them as a particular class of dynamical systems. Let us start by adapting Definition 3.1. The model is considered for discrete event systems and thus, they have discrete states and a transition period considered constant and known (μ). Events have the *timed/untimed* form presented in Section 3.3. To make the definition simpler only timed events will be considered. The untimed events, as discussed above, will have time guards of $t_1 = 0$ and $t_2 = \infty$.

Instead of only one output function we will consider a set of output functions, indexed over a set J, of indexes. The functions are time independent, depending only on the state the systems are in at that moment. The set of admissible input functions is included in the set $(E \times T)^*$. Thus, Ω will not be represented anymore in the system definition. The system described by Definition 3.1 reacts instantaneously to inputs. Our system has a delay between the time the event appears at the input and the moment when the system will change its state. Over an interval $[t_0, t_1)$ the system can process at most $int(t_1 - t_0/\mu)$ events. The function φ was replaced by the function δ.

Definition 3.3 *A **DDES** is a tuple $A = (T, Q, E, Y, \delta, \beta)$ where:*

- $T = \mathrm{R}^+$ *is a set of **time values** and μ is a time interval called **transition time**.*

- *E is a set of* ***events*** *and $I = E \times T$ is called the input alphabet.*
- *Q is a set of symbols called* ***states.***
- *$Y = (Y_j \mid j \in J)$ is an indexed set of* ***output values.***
- *δ is a function $\delta : T \times Q \times I \longrightarrow Q$, called the* ***next state function*** *defined as:*

$$\forall t, t_d \in T, t \geq t_d$$

$$q(t+\mu) = \delta(t, q, (e_{t_1}^{t_2}, t_d)) = \begin{cases} q_1 & \text{if } t \in [t_d + t_1, t_d + t_2) \\ q & \text{otherwise} \end{cases}$$

where $e_{t_1}^{t_2}) \in E$ is the event, t_d is the moment when the event appears and $t \geq t_d$ is the moment when the event is processed.

- *β is a set of total functions $\beta = \{\beta_j | \beta_j : Q \longrightarrow Y_j\}$, called* ***output functions.***

■

An event e is said to be **enabled** if it has an outcome on the system. Untimed events are always enabled, as opposed to their timed counterparts that are enabled only within a certain time interval. The null event (ε) although it is untimed is never enabled and does not have any outcome for the system. If an event is enabled, a system $\mathcal{S}$ in state $q \in Q$ will change its state to $q_1 \in Q$ after the interval μ has elapsed.

A *DES* is in general *time variant.* It can be easily proved that Definition 3.3 satisfies that property. Let us choose an event (e, t_1, t_2) and the value of τ from Definition 3.2 to be greater than t_2. Then $\forall t \in [t_1, t_2)$ the value of $t + \tau > t_2$ and thus, $\delta(t, q, e) \neq \delta(t + \tau, q, e)$. Therefore, by Definition 3.2 the system is *time variant.* If, on the other hand, we restrict the class of acceptable events to the class of untimed events, the system becomes *time invariant.* In this case time becomes unimportant and the DDES is a modified form of a classical automaton, with the time and transition period defined over the real numbers.

A DES as described by the above definition will start functioning at time $\tau \in T$ in global time and 0 in local time. The state that the DDES is in, at that moment, is called *initial state.* It should be noted that depending on the chosen initial moment the initial state doesn't have to

be the same. We will restrict the definition by considering only systems that have a single initial state regardless of the time moment when the system starts.

If the DDES has a terminating execution, defining a set of states that describe a valid end is useful. These states are called *final* and the set that they form is called the *set of final states.* As it is known, most of the real-time systems are *nonterminating.* Therefore, termination means that the system crashed. For these systems the set of final states should be empty. If the sets E, Q, Y are finite, the DDES will be called *finite DDES.*

Definition 3.4 *A* discrete event automata (DEA) *is a DDES A together with an initial state* q_0 *and a set of final states* $F \subseteq Q$. $B = (T, Q, E, Y, \delta, \Gamma, q_0, F)$. ■

Our main interest is the reactive behavior of the systems. Hence, in the sequel we will consider the set F to be empty and omit it from the system description. Reactive behavior also implies that the input function $\omega : T \longrightarrow E$ is defined over an interval σ_t, because our machine starts at $t_0 = 0$. As discussed above the number of points where the function is defined has to be *countable.*

The DEA is a generalization of Arbib's [6, 38] notion of automata. If we further restrict our definition to a natural time set and consider only untimed events we get his version of automata.

The *run map* [2] of a DEA is defined over the set $(E \times T)^*$ by a function $\rho : (E \times T)^* \longrightarrow Q$. The function is evidently partially defined, as the set of admissible input functions accepts only strings that satisfy the time relation over T. ρ is recursively defined as:

- $\rho(\Lambda) = q_0$
- $\rho(C_n(e_{t_1}^{t_2}, t_d)) = q_n)$
- $\rho(C_n(e_{t_1}^{t_2}, t_d)(e_{t_{11}}^{t_{21}}, t_{dn})) = q_{n+1} = \delta(t, q_n, (e_{t_{11}}^{t_{21}}, t_{dn})$

The strings $s \in (E \times T)^*$ that appear as inputs to a system are called *input runs.* The set of admissible *input runs* gives the set of such strings that do not generate Zeno behaviors.

Related with the *run map* is the behavior $\gamma \in \Gamma$. In the case of DEA, γ is a set of functions defined as: $\gamma_j = \beta_j \circ \rho$. Thus, γ_j is defined over $(E \times T)^* \longrightarrow Y_j$ and is a partial function, as is also the case for ρ.

The main point of interest, as stated at the beginning of the chapter, is the way we can combine simpler structures in order to obtain more complicated ones. Another way of obtaining new

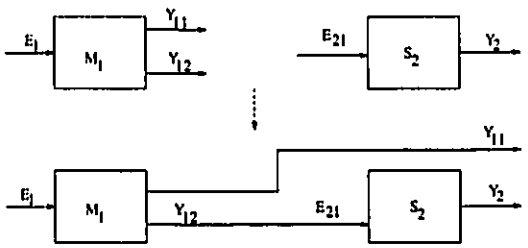


Figure 3.5: Series composition of DEA's

systems from known ones is by transforming them. The following definition gives the restrictions needed in order to build a morphism between two DEAs.

Definition 3.5 *Let* $M_1 = (T, E_1, Q_1, Y_1, \delta_1, \beta_1, q_{01})$ *and* $M_2 = (T, E_2, Q_2, Y_2, \delta_2, \beta_2, q_{02})$ *be two DEAs. A* ***morphism*** *between* M_1 *and* M_2 *is a set of functions* $\{tm : T \longrightarrow T, h : Q_1 \longrightarrow Q_2, ev : E_1 \longrightarrow E_2\} \cup \{g_j : Y_{1j} \longrightarrow Y_{2j} \mid \forall j \in J\}$ *satisfying the following conditions:* $f(\tau_1) = \tau_2$, $f(\mu_1) = \mu_2$, $h(q_{01}) = q_{02}$, $h(\delta_1(t, q_1, (e_1, t_d))) = \delta_2(tm(t), h(q_1), (ev(e_1), tm(t_d)))$ *and* $\forall j \in J, g_j(\beta_{1j}(q_1)) = \beta_{2j}(h(q_1))$. ■

Using three types of compositions,*series*, *parallel* and *feedback*, we can obtain complex systems from simpler ones. Although used a lot by system theory, composition is weakly represented in the domain of formal models. Most of the authors seem concerned with the monolithic description of the system and except for parallel composition [5, 79, 37, 55, 9] none of the other is considered. Parallel composition provides an fairly accurate description of behavior for *non interacting* systems functioning simultaneously. For systems that also interact with each other more sophisticated constructions are needed. Some even define [70] *parallel compositions* although the system is a series construction with feedback. Following classical work in system theory [60] and automata theory [6] let us define the *series*, *parallel* and *feedback* composition.

In the *series* configuration (Figure 3.5) two automata are linked so that outputs from one automaton are connected to inputs of the other automaton. Of course the connection can be made if and only if the set $Y_{12} \subset E_{21}$. Without this connection compatibility the second system will not make any transitions. In the composed system both components start at the same time τ and thus, the set T is the same on both systems. Furthermore the output of the first system has to be event driven in order to have the same structure as the other events. Without this, the

output would generate an uncountable number of events and this would be a Zeno behavior.

The sets of admissible inputs are, usually, different between DEAs as a result of different transitions times and input queue lengths. To have a resulting system with non Zeno behaviors, it is necessary to impose restrictions on the admissible input runs for the composed system. The admissible runs are determined by the transition time. To obtain them for the composed system let us compare the transition times for the parts. If $\mu_1 \geq \mu_2$ then the actions generated by the first system are at least μ_1 apart and this second system will synchronously process them as they appear. Hence, the set of acceptable inputs for M_1 is the set of acceptable inputs for the composed system. If $\mu_1 < \mu_2$ the set of admissible inputs for the composed system should be reduced such that the arrival rate is smaller than $1/\mu_2$. Therefore the arrival rate for the composed system should be: $1/max(\mu_1, \mu_2)$.

In the sequel only the case when $\mu_1 > \mu_2$ is considered. The transition time of the composed system is then $\mu = \mu_1 + \mu_2$. The above equation states that the outputs are considered to change only after the effect of an input event had propagated through the system.

Definition 3.6 *Let* $M_1 = (T, E_1, Q_1, Y_1, \delta_1, \beta_1, q_{01})$ *and*

$$M_2 = (T, E_2, Q_2, Y_2, \delta_2, \beta_2, q_{02})$$

be two DEA's, where $Y_1 = (Y_{11}, Y_{12})$, $\beta = (\beta_{11}, \beta_{12})$. *The series composition of* M_1 *and* M_2 *denoted by* $(M_1 \odot M_2)$ *is a system*

$$M = M_1 \odot M_2 = (T, E, Q, Y, \delta, \beta, q_0)$$

where

- $Q = Q_1 \times Q_2$ *is the state set.*
- $E = E_1$ *is the event set and* $I = E_1 \times T$ *is the input alphabet.*
- $Y = (Y_{11}, Y_2)$ *is the output set.*
- $\delta : T \times Q \times I \longrightarrow Q$ *is the next state function.*

$$\delta(t, (q_1, q_2), (e_1, t_d)) = (\delta_1(t, q_1, (e_1, t_d)), \delta_2(t, q_2, (\beta_{11}(q_1), t_d)))$$

- $\beta = (\beta_{11}, \beta_2)$ *is the set of output functions.*
- $q_0 = (q_{01}, q_{02})$ *is the* initial state.

■

Let us consider the case when several inputs are applied to the same system. There are two possible points of views for this kind of structured inputs. The first one considers the events to be synchronized [36].

To explain this let us consider two DEA's M_1 and M_2 working in parallel. If E_1 is the set of admissible events for M_1 and E_2 for M_2, the set of events for the resulting system is $E' = \{(e_1, \varepsilon) \mid e_1 \in E_1\} \cup \{(\varepsilon, e_2) \mid e_2 \in E_2\} \cup \{(e_1, e_2) \mid e_1 \in E_1, e_2 \in E_2\}$. The set of events for the parallel composition denotes that the events can appear at the same moment in time. Therefore they can be synchronized, or appear at different moments and be desynchronized. In each case the composed events are the doubles (e, ε) or (ε, e) where ε represents a "non-event". This point of view considers all the events queued in a single unit and processed synchronously. If $I_j = E_j \times T$ is the input alphabet for one input then because $T \times T \times ...T \cong T$ the alphabet of the composed inputs is $E_1 \times E_2..E_n \times T$.

A second point of view is to consider the inputs to be independently processed. Considering the inputs independent though, contradicts our intuition that systems behave like monoliths.

In the *parallel* configuration the set of events E of the resulting machine is a subset of the Cartesian product of the sets of events of the composing machines $E \subset E_1 \times E_2$. As with the *serial connection*, both machines have the same time reference, with the same initial moment. This ensures that the elements of the Cartesian product are valid events for the composed machine.

Definition 3.7 *Let* $M_1 = (T, E_1, Q_1, Y_1, \delta_1, \beta_1, q_{01})$ *and*

$$M_2 = (T, E_2, Q_2, Y_2, \delta_2, \beta_2, F_2, q_{02})$$

be two DEA's (Figure 3.6), with $E_1 = (E_{11})$ *and* $E_2 = (E_{21} \times E_{22})$. *The parallel composition of* M_1 *and* M_2 *denoted by* $(M_1 \oplus M_2)$ *is a system*

$$M = M_1 \oplus M_2 = (T, E, Q, Y, \delta, \beta, q_0)$$

where

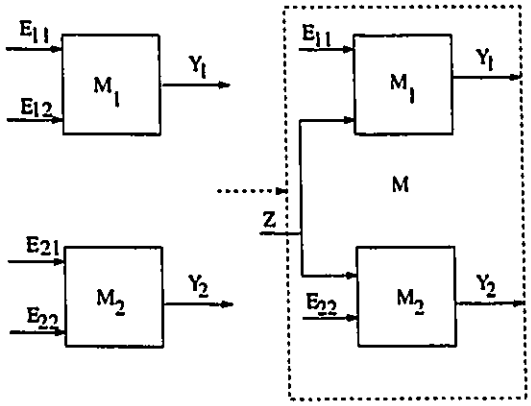


Figure 3.6: Parallel composition of DEA's

- T *is the set of time values.*
- $Q = Q_1 \times Q_2$ *is the set of states and* $q_0 = (q_{01}, q_{02})$.
- $E = (E_{11} \times Z \times E_{22})$, *with* $Z \subseteq E_{12} \cap E_{21}$, *is the set of events and* $I = E_{11} \times Z \times E_{22} \times T$ *is the input alphabet.*
- $Y = (Y_1, Y_2)$ *is the set of output values.*
- $\delta : T \times \times Q \times I \longrightarrow Q$ *is the next state function, defined by:*

$$\delta(t,(q_1,q_2),(e_1,z,e_2,t_d)) = (\delta_1(t,q_1,(e_1,z,t_d)),\delta_2(tq_2,(e_2,z,t_d)))$$

,

- $\beta = \{\beta_1, \beta_2\}$ *is the set of output functions.*

■

The transition time of the composed system is equal to $max(\mu_1, \mu_2)$.

In the *feedback* connection some of the system's outputs are connected with its inputs thus forming another system. The feedback system can make more than one transition for a certain input, because the output that is generated will be another event for the system. The DEA of the Definition 3.4 makes only one transition after it receives an event. In this case we define the transition of the feedback system to take place only when it reaches a state that will not change

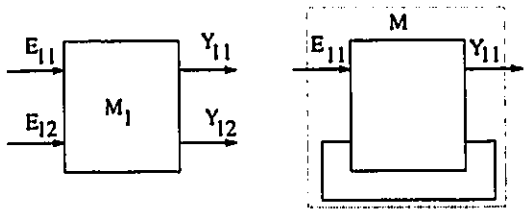


Figure 3.7: Feedback composition of DEA's

unless it receives an input. Evidently the system can make several internal transitions in order to enter such a state. The set of admissible input functions for the feedback system is only a subset of that of the original system. This is due to the increased processing time.

Definition 3.8 *Let* $M_1 = (T, E_1, Q_1, Y_1, \delta_1, \beta_1, F_1, q_{01})$ *be a DEA (Figure 3.7), with* $E_1 = (E_{11} \times E_{12})$ *and* $Y_1 = (Y_{11}, Y_{12})$. *The* ***feedback*** *connection of* M_1 *is a system*

$$M = (T, E, Q, Y, \delta, \beta, q_0, F)$$

, *where:*

- T *is the set of time values.*
- $Q = Q_1$ *is the set of states, with* $q_0 = q_{01}$.
- $E = E_{12}$ *is the set of events and* $I = E \times T$ *is the input alphabet.*
- $Y = Y_{12}$ *is the set of outputs.*
- $\beta = \{\beta_1\}$ *is the set of output functions.*
- $\delta : T \times Q \times I \longrightarrow Q$ *is the next state function:*

$$q_1(t + n * \mu) = \delta(t, q, (e_{11}, t_d)) = \delta_1(...(\delta_1(t, q, (e_{11}, t_d))))$$

.

■

Due to their nature, some feedback systems can have an infinite transition time. This is because the system will make internal transitions forever. This suggests that the class of all DEA's is not closed under the feedback operation. Thus, such an operation has to be applied with caution.

3.5 DEAs as algebras

The relation between automata theory and algebra, an idea that seems almost natural now, has been first pointed out by Büchi [10], that studied automata as *unary algebras*. By reducing the class of abstract automata to a unary algebra, only one set of the structure can be studied and therefore simplifications have to be made. Pointing out the problem Büchi [10] also noted that means of a more general approach can study the same structures. He proposed *multi sorted algebra* which, presents the advantage of allowing an indexed carrier for the algebra. Therefore, the totality condition imposed for the operators of an algebra can be satisfied over subsets of the carrier. Noticing the relation of universal algebra with another mathematical field of study, category theory, automata theorists and mathematicians tried to link the two fields together. Manes and Arbib [7] have started a categorical approach by proving that the class of automata is a category and also, that the automaton itself can be viewed as a category. In this case automata properties are derived using categorical concepts. Adamek [2] used an interesting variant to the categorical approach, by studying automata as *functors* between categories.

The algebraic methods of study are at help when one tries to make the transition from the automata as models of computations to logic theories that describe them. The relation of universal algebra with logic, by the means of model theory, means that a link between logical theories and the abstract models can be made. This is the reason of approaching the DDES/DEA and therefore distributes systems through an algebraic theory.

Let us now consider the **DDES/DEA** framework in the light of universal algebra. The multi set nature of the framework makes it a natural candidate for study using *multi sorted* algebra.

Let $M = (T, Q, E, Y, \delta, \beta)$ be a **DDES**. The condition $t \leq t_d$ makes δ partial. To make a **DDES** suitable for study using the universal algebra framework, we need to make δ a total function. This can be done by considering an extension of δ such that, for all the doubles (t, t_d) where the function is undefined, $\delta(t, q, (e, t_d)) = q$. In the remainder of the section δ will be considered to be the extended *next state function* $\bar{\delta}$. As previously stated, for **DEAs** only the reactive case with $F = \phi$ is considered. Hence, the set F does not appear explicitly. As expected both *DDES* and *DEA*, with the extended transition function, have the structure of universal algebras as proven by the following lemmas.

Lemma 3.1 *A DDES is a multi sorted algebra.* ■

Proof: Let $M = (T, Q, E, Y, \delta, \beta)$ be a **DDES**. Let us attach a sort to each of the sets. Hence, we'll consider T of having sort s_1 and Q of having sort s_2. The set of output values is made of several subsets: $Y = (Y_1, .., Y_j)$. Each Y_j is considered to have sort j. Finally consider E of having sort i. Thus, the set S of sorts is composed of: $\{s_1, s_2, i\} \cup \{j | midj \in J\}$. Over S consider the signature $\Sigma = (\delta, q_0, \beta_j \mid j \in J)$ with the following arities: $ar(\delta) = (s_1 s_1 s_2 i, s_2)$, $ar(\beta_j) = (s_2, j)$.

The if we consider the set $C = (T_{s_1}, Q_{s_2}, E_i, \{Y_j \mid j \in J\})$ to be the carrier and the functions δ and $\beta_j \in \beta$ from M to be realizations of the operation symbols from Σ, by Definition 2.10, $A = (C, \delta, \{\beta_j \mid j \in J\})$ is a multi sorted algebra. Therefore, M is a *multi sorted algebra.*

■

Lemma 3.2 *A **DEA** is a multi sorted algebra.* ■

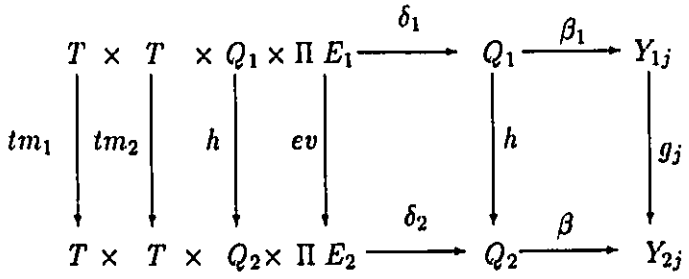
Proof: Let $M = (T, Q, E, Y, \delta, \beta, q_0)$ be a **DEA**. As in the case of **DDES** let us consider the set $C = \{T_{s_1}, Q_{s_2}, E_i, \{Y_j \mid j \in J\})$ with the arities specified by the indexes.

If we consider q_0 to be an operation symbol, then we obtain the signature $\Sigma = (\delta, \{\beta_j \mid j \in J\}, q_0,)$ with the following arities: $ar(\delta) = (s_1 s_1 s_2 i, s_2)$, $ar(\beta_j) = (s_2, j)$, $ar(q_0) = s_2$. Hence, the set C with the operation defined by σ form, according to Definition 2.10, is a multi sorted algebra. Therefore, M is a *multi sorted algebra.*

■

DEAs are therefore *multi sorted* algebras. In the universe of algebras homomorphisms make "links" between algebras. Thus, it is interesting to notice that the morphisms between DEA's are homomorphisms in the universe of multi sorted algebras.

Corollary 3.1 *A morphism between two DEA's is a homomorphism between the underlying algebras.* ■



Proof: Let $M_1 = (T, E_1, Q_1, Y_1, \delta_1, \beta_1, q_{01})$ and $M_2 = (T, E_2, Q_2, Y_2, \delta_2, \beta_2, q_{02})$ be two DEAs with Y_1 and Y_2 defined over the same J. By Lemma 3.2 M_1 and M_2 are algebras over the same signature $\Sigma = (\delta, \{\beta_j \mid j \in J\}, q_0)$.

The set of functions $\{tm_1 : T \longrightarrow T, tm_2 : T \longrightarrow T, h : Q_1 \longrightarrow Q_2, ev : E_1 \longrightarrow E_2\} \cup \{g_j : Y_{1j} \longrightarrow Y_{2j} \mid \forall j \in J\}$ satisfying the morphism conditions of Definition 3.5 also satisfy Definition 2.11 conditions for multi sorted homomorphism.

Over the signature Σ, we have:

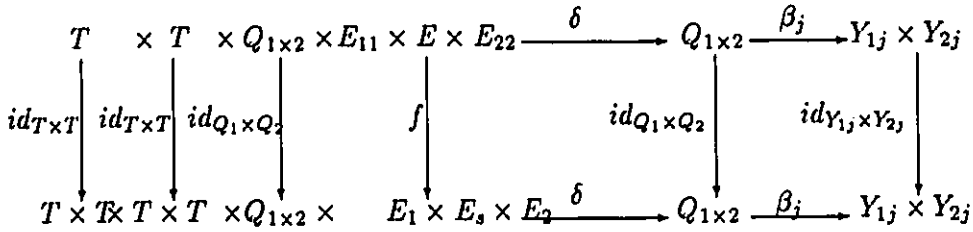
- $h(q_{01}) = q_{02}$ by hypothesis.
- $\forall f_1 \in F_1 \exists f_2 \in F_2$ such that $h(f_1) = f_2$ by hypothesis.
- $h(\delta_1(t_1, t_2, q_1, e_1)) = \delta_2(tm(t_1), tm(t_2), h(q_1), ev(e_1))$ and
- $\forall j \in J, g_j(\beta_{1j}(q_1)) = \beta_{2j}(h(q_1))$.

Therefore, by Definition 2.11 $[tm_1, tm_2, h, ev, (g_j \mid j \in J)]$ is a Σ-homomorphism.

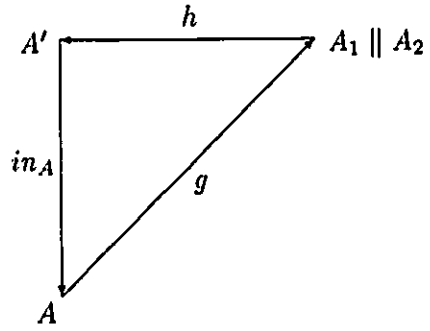
■

The concept of **direct product** of algebras captures the behavior of parallel working automata. Let $A_1 = (T, E_1, Q_1, Y_1, \delta_1, \beta_1, q_{01})$ and $A_2 = (T, E_2, Q_2, Y_2, \delta_2, \beta_2, q_{02})$ be two DEAs and B their parallel composition with respect to Definition 3.7. The direct product $A = A_1 \times A_2$ is the algebra $A = (T \times T, E_1 \times E_2, Q_1 \times Q_2, Y_1 \times Y_2, \delta, \beta, (q_{01}, q_{02}))$, where the functions δ and $\beta \in \beta$ are defined point wise. Let us study the relation between the Cartesian product of two algebras and the *parallel* composition of the DEA's.

More precisely, let us first consider the parallel connection of two DEAs as in Figure 3.6. Let $B = (T, E, Q, Y, \delta, \beta, q_0)$ be the multi sorted algebra obtained from the result of the parallel composition of the A_1 and A_2. For the parallel connection to have meaning, we have to impose the condition $E = E_{12} \cap E_{21}$. By considering the subset $E_s = \{(e, e) \mid e \in E\} \subset E_{12} \times E_{21}$ we can build a sub algebra A' of A (the direct product of A_1 and A_2) formed with the same sets as A except for the event set, where we will take only a subset $E_v = E_{11} \times E_s \times E_{22} \subset E_{11} \times E_{12} \times E_{21} \times E_{22}$. Furthermore let us consider the following function $f : E_{11} \times E \times E_{22} \longrightarrow E_{11} \times E_s \times E_{22}$ defined as $f(e_{11}, e, e_{22}) = (e_{11}, e, e, e_{22})$. As previously discussed for the time set $T \times T \cong T$. Let us consider the function $id_{T \times T} : T \longrightarrow T \times T$, defined as: $id_{T \times T}(t) = (t, t)$. Hence, $h = [id_{T \times T}, id_{Q_1 \times Q_2}, f, id_{Y_j}]$ is a homomorphism between B, the algebra representing the parallel connection, and A'. Let us denote $E_1 = E_{11} \times E_{12}$, $E_2 = E_{21} \times E_{22}$ and $Q_{1\times 2} = Q_1 \times Q_2$.



However, A' is a sub algebra of A. Therefore, there exists a homomorphism $in_A : A' \longrightarrow A$. By applying the Universal Property of Algebra we obtain a homomorphism from B to A. Hence, there exits a homomorphism from the underlying algebra of the parallel connection and the Cartesian product of A_1 and A_2.



A characterization from $A_1 \times A_2$ to B is also possible. In this case we need a function $tm : T \times T' \longrightarrow T$ and a function $g : E_{11} \times E_{12} \times E_{21} \times E_{22} \longrightarrow E_{11} \times E \times E_{22}$. The function tm can be defined as

$$tm(t_1, t_2) = \begin{cases} t & \text{if } t_1 = t_2 \\ 0 & \text{otherwise} \end{cases}$$

and the function g by the following description:

$$g(e_{11}, e_{12}, e_{21}, e_{22}) = \begin{cases} (e_{11}, e, e_{22}) & \text{if } e_{12} = e_{21} \\ (e_{11}, \varepsilon, e_{22}) & \text{otherwise} \end{cases}$$

Thus, $k = [tm, tm, g, id_{Q_1 \times Q_2}, id_{Y_j}]$ is a homomorphism from $A_1 \times A_2$ to B.

3.6 A Model for Sequential Processes

This section examines the structure of computer processes, using the formal model that was built in the previous paragraphs. The abstraction process uses a "reversed engineering" method

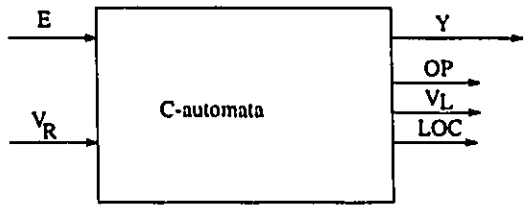


Figure 3.8: C-automata

by trying to formalize a concept derived from a human made tool to produce other tools. Our technological perception of the "computational process" biases this type of abstraction.

Multitasking systems appeared after the sequential ones and tried to mimic their behavior, by creating the illusion of multiple parallel processes executed on a single physical unit. Thus the notion of "sequential process" is crucial to the formalization process.

In this thesis sequential processes are considered to be collections composed of two entities that interact with each other. One entity ensures the dynamics of the process and therefore, acts as a *controller*, while the other one acts as a *memory* that stores variables for the controller. Let us examine both parts from the perspective of the **DDES/DEA** model developed in the previous sections.

The *control* part of the process will be considered to be a DEA. The abstraction can be made because the controller has discrete states and accepts discrete events.

Let us consider a particular kind of **DEA** with the structure of inputs and outputs depicted in Figure 3.8. Let us call this type of system a *C-automaton*. The set of inputs for the system is structured, as visible in Figure 3.8. This is because some inputs (V_R) are needed to interact with the *memory* part of the process, as the others are needed for external interaction E. The set of outputs is also composed of several subsets. Part of the outputs are inputs in the memory subsystem (OP, V_l, LOC) and the rest are the outputs of the process. Thus, a C-automaton is a DEA with the following structure:

$$A_c = (T, \{E, V_R\}, Q, \{LOC, OP, V_L\}, Y, F, \delta, \{\beta_1, \beta_2\}, q_0)$$

where

- $E_c = V_R \times E$ is a set of events.

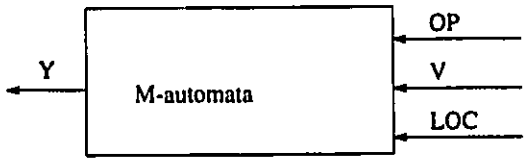


Figure 3.9: M-automata

- Q is a set of states. $q_0 \in Q$ is the initial state.
- $\beta = V_L \times OP \times LOC \times Y$ is the a set of output events.
- $\delta : T \times T \times Q \times E_c \longrightarrow Q$ is the next state function, $\delta(q, e_c) = q_{next}$
- $\beta = (\beta_1, \beta_2), \beta_1 : Q \longrightarrow Y, \beta_2 : Q \longrightarrow OP \times V_L \times LOC$ is the output function.

The system has two output functions, β_1 and β_2, to make the distinction between the process output and the commands for the memory.

To be able to make a suitable abstraction for the memory we must analyze how a real memory behaves. The term can mean many things, but here we refer to a device that can store and access information by direct addressing. In the technology used for most of todays computers a memory system is just a conglomerate of units called *memory cells*. A *memory cell* is a place where a quantity of information can be stored at times and retrieved afterwards. The number of different symbols that can be stored depends on the memory and is always finite. All the events that a memory receives are untimed. Furthermore the memory system is time invariant and thus the time set can be omitted from the definition of the system. By considering the stored symbols as the *states* of the memory we can abstract it as a DEA with the following structure: $(Q, Q, Q, \delta, \{\beta\}, F, q_0)$. The notion of final state does not make sense for a storage facility. We will consider $F = \phi$ and omit it from subsequent references.

The model can be expanded by considering several memory units running in parallel. If we use parallel composition we can obtain a memory with n cells: $(Q^n, Q^n, Q^n, \delta, \beta, q_0^n) = (Q^n, \delta, \beta, q_0^n)$ where $\delta : Q^n \times Q^n \longrightarrow Q^n$ and $\beta : Q^n \longrightarrow Q^n$. Let $\beta_m : Q^n \times E \longrightarrow Q^n$.

Let $E_m = \{(-, -, ..., q_i, ..., -) \mid q_i \in Q\} \subset Q^n$ such that only one memory cell receives an event at a certain time moment. And let OP, V_L, LOC be three sets of symbols such that

$OP = \{R, WR\}, V_L \subset Q, LOC \subset \mathbb{N}$. Let $f : OP \times V_L \times LOC \longrightarrow E_m$ be a function such that

$$f(x,y,z) = \begin{cases} \lambda & \text{if } x = R \\ (\underbrace{-,-,...,}_{z} y, ...) & \text{if } x = WR \end{cases}$$

and $\delta_M : OP \times V_L \times LOC, \delta_M = \delta \circ f$. We can now define a model for the memory that we'll call *M-automaton.*

Definition 3.9 *An M-automaton is DEA,* $A_M = (\{OP, V_L, LOC\}, Q^n, Q, \delta, \beta, q_0^n)$ *where*

- *E is the set of events $E = OP \times V_L \times LOC$, where:*
  - *$OP = \{R, W\}$ is a set of symbols called operations;*
  - *V_L, LOC are finite sets of symbols called values and locations.*
- *δ is the next state function $\delta : Q^n \times E \longrightarrow Q^n$*
- *β is the output function $\beta : Q^n \times LOC \longrightarrow Q$*
- *q_0^n is the initial state: $q_0^n \in Q^n$*

■

We can now present a formal definition of the notion of process. As previously said a process is considered to be composed of two parts that interact with each other: a control part that is described by a *C-automaton* and a memory part that is formalized by an *M-automaton.* Let us consider the two automata to be connected as depicted in Figure 3.10. This type of connection can be expressed as a combination of serial and feedback connections: $P = \mathcal{F}(M \odot C)$. Let us call the resulting system a *P-automaton.*

Definition 3.10 *Let*

$$A_c = (T, \{E_c, V_R\}, Q_c, \{LOC, OP, V_L\}, Y_c, F_c, \delta_c, \{\beta_{1c}, \beta_{2c}\}, q_{0c})$$

be a C-automaton and

$$A_M = (\{OP, V_L, LOC\}, Q_M^n, Q_M, \delta_M, \beta_M, q_{0M}^n)$$

be a M-automaton. A ***P-automaton*** *is a septuple*

$$A_p = \mathcal{F}(A_C \odot A_M) = (T, E_p, Q_p, Y_p, \delta_p, \beta_p, q_{0p}, F_p, \{LOC, OP, V_L, V_R\})$$

resulting by the composition of the A_C and A_M where

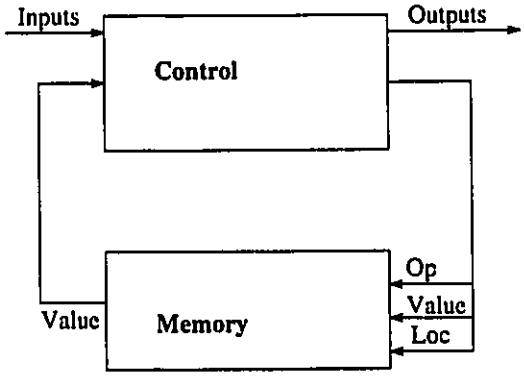


Figure 3.10: Process abstraction

- $Q_p = Q_C \times Q_M^n$ *is the set of states, and* $q_{0p} = (q_{0C}, q_{0M}^n)$ *is the initial state.*
- β_p *is the output function.* $\beta_p((q_1, q_2)) = (\beta_{1C}(q))$
- $F_p = \{(f_C, q) | f_c \in F, q \in Q_M\} \subseteq Q_p$ *is the set of final states.*
- $E_p = E_c$ *is the set of (external) events.*
- δ_p *is the next state function,*

$$\delta : T \times T \times q_p \times E_p \longrightarrow Q_p$$

,

$$\delta(t, t, (q_1, q_2), e) = (\delta_C(t, q, ((e, v_R), t_e)), \delta_M(q_2, (op, v_L, loc)))$$

$$= (\delta_C(t, q_1, ((e, \beta_M(q_2), t))), \delta_M(q_2, \beta_{2C}(q)))$$

■

The state of a sequential process is given by the values of its variables plus the state of the processor. For a *P-automaton* the state is the state of the control part plus the state of all the memory locations (values found in the memory cells). Therefore, using the above definition, the state of the process corresponds with the state of the *P-automaton*. This model is related with the *fair transition systems (FTS)* [56]. Let us consider the state set of *P-automata* $Q_p = Q_c \times Q_M^n$. A variable $v \in Q_p$ is a tuple $v = (v_c, v_M^1, ..., v_M^n)$. The composing elements of this variable are *state*

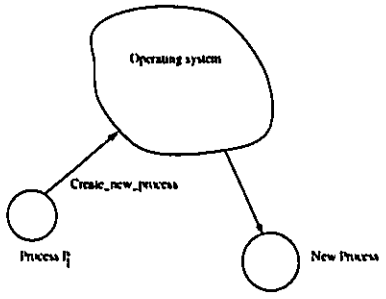


Figure 3.11: Creation of new processes

variables of *FTS*. The transition τ is the same as our function δ and the *initial condition* from *FST* is our q_0. In [56], the authors add two more classes of restrictions to their models called *justice* and *fairness* in order to insure proper operation of the system.

From Definition 3.10 we can derive the definition of a reactive process.

Definition 3.11 *A **reactive process** is a sequential process that has an empty set of final states.*

■

A typical concurrent system has a variable number of processes during its life time that are created and killed at the request of other processes. The program that does this work is called *operating system (OS)*. After a process is created the relation between the processes that asked for the service and the new process is determined by the specifics of the system.

From the point of view of process creation, the *operating system* can be abstracted by considering an entity that contains a set of *P-automaton*. This is the set of processes that the system can create. Furthermore, an automaton that has a special type of next state function is needed to start a new process when needed. When the *OS* automata receives an event requesting a new process it will make not only an internal transition, but will also start another *P-automata*. Let us call the abstraction of the *OS* an *S-automaton*.

Definition 3.12 *An S-automata is septuple*

$$A_{s_i} = (T, E, Q, \delta, Y, \mathcal{A}_P, \beta, q_0)$$

where:

- *T is a set of time values,*
- *E is a set of events,*
- *Q is a set of states with $q_0 \in Q$ the initial state,*
- *$\mathcal{A}_P$ is a set of P-automata and Y a set of output values,*
- *$\delta : T \times T \times Q \times E \longrightarrow Q$ is the next state function,*
- *β is the output function, $\beta : Q \longrightarrow Y \times \mathcal{A}_P$,*

■

An *S-automaton* is thus, a DES those set of output values is a set of *P-automata*. The *S-automaton* can output two types of actions: one is the same as in the case of DEAs, a simple output, while the other is a new process. The set T is considered to be the set of real numbers as in all the other formalization discussed before. The new process will start its work in its initial state. One major difference between this abstraction and a real system is that once it starts a new process the operating system losses control over it and cannot stop it. Although this might seem like a major drawback, in my opinion it is not. To be more precise, once the process is started, if the process has a reactive behavior it should not stop.

3.7 Distributed systems

The next logical step in the abstraction process is to present a model for distributed systems ($\mathcal{DS}$). The preamble of this chapter presents our concrete view of a distributed system: a double, composed of a set of processes and a set of communication channels. By a communication channel we understand a module that insures the transport of information between two entities. This concept includes the communication protocol and the potential delays between transmition and reception.

In the light of our previous discussion on the abstraction of sequential processes, there are two possibilities in abstracting the process set: either as a set of *P-automata* in which case the system doesn't have any variability in the number of running processes, or as a set of *S-automata* in which case the number of running processes in a $\mathcal{DS}$ is variable. In this thesis we will consider

the distributed system to have a fixed number of processes since its inception. Therefore, the set of processes will be a set of *P-automata.*

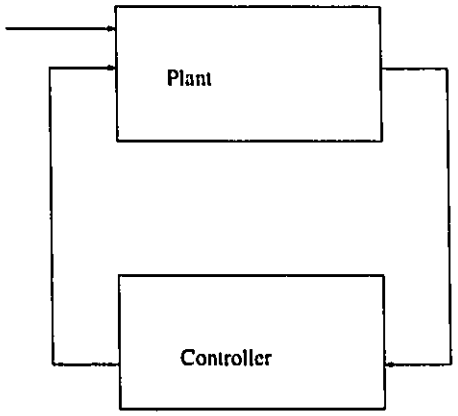


Figure 3.12: The system model of the process set

Communication protocols tend to be fairly simple, as compared with the process part of any system. They can usually be described in terms of finite state machines. Since our concept of DEA is more general it is enough for describing the communication protocols of practical interest. Furthermore, using μ (processing time) one can impose delay restrictions in the message delivery. Thus, DEA is a powerful enough concept for our notion of communication channels.

Let us summarize the above discussion in a definition:

Definition 3.13 *A distributed system $\mathcal{DS}$ is a double $(\mathcal{P},\mathcal{C})$ where $\mathcal{P}$ is a set of* ***processes*** *and $\mathcal{C}$ is a set of* ***communication channels.*** *The set $\mathcal{P}$ is a set of P-automata and the set $\mathcal{C}$ is a set of DEAs.* ■

Processes communicate with each other through communication channels that can be either unidirectional or bidirectional. We can also view the latter case as a sum of two unidirectional communication channels that send information in opposite directions. Here, a process is a P-automaton and thus, the connection of the process to a unidirectional communication channel is a series connection between two DEAs. If we consider two processes connected though an unidirectional channel, the sending connection is a *series* connection between P_1, a process, and C, a communication channel. Therefore, it can be denoted by $P_1 \odot C$. The receiving side is also

a series connection, in this case between C, a communication channel, and P_2, a process, and it is denoted by $C \odot P_2$.

The sending side is realized by connecting some outputs of a *P-automaton* to the inputs of the communication channel. The receiving side is made by connecting the outputs of the communication channel to the inputs of the process.

Therefore, the fact that the process P_1 sends a message to the process P_2 through channel C is denoted by $P_1 \odot C \odot P_2 = P_1 \rhd_C P_2$. If there is also a connection in the opposite direction it will be denoted by $P_1 \lhd_C P_2$. For the bidirectional connection this becomes $P_1 \bowtie_C P_2$.

Another aspect in the model is the structure of the process set. To make the system usable some form of control is needed. In our case we consider the set of processes to be partitioned into two disjunct subsets:

- One that contains the processes that have to be controlled. This set is called *plant* by control theorists or the set of *clients* by computer engineers. Let us denote this set by P_S. Usually the processes that form this set interact with the environment and as a result of this interaction make requests to the controller.

- The processes that control the *plant* form the other subset, called the *system controller (C_S) set* or *server set*. The relation between P_S,C_S and $\mathcal{P}$ can be summarized in the following equation: $P_S = \mathcal{P} - C_S$ (Figure 3.12. The controller can be composed of several processes that interact with each other in the manner described above.

By reducing the set C_S to one element, Lin's [37] theory for controller generation can be applied. His method consists of several steps. First the *plant* is specified. On the plant, which is usually composed of several modules, several restrictions are imposed. Using these restrictions, the events that lead to the unwanted states can be detected. These events will be subsequently inhibited by the controller. The important part in the method is that it was proved that the generation of the controller also keeps place for *verification*. It has to be said that Lin did not consider outputs in his model. His procedure makes a so called "synchronization of events". This is equivalent with specifying the relation between the inputs and outputs of the *plant* and *controller*. Since our model has both inputs and outputs, synchronization in Lin's way is not needed for our model.

Chapter 4

Temporal Logic

Temporal logic, as any branch of mathematical logic, tries to use the power of formal reasoning to make deductions in a given axiomatic system. In this chapter a variant of the linear time temporal logic will be introduced. Subsequently, a semantic relation with the abstract models developed in the previous chapter will be presented.

The present logical framework is based Manna and Pnueli's temporal logic described in [55]. As with all the other linear time temporal logics, the interpretation of formulas in their logic is done over a linear sequence of time moments. Temporal logic was first applied to the description of DES by Ostroff [70]. He builds formulas over variables that are part of *extended state machines*. Lin and Ionescu [51] extended temporal logic used for DES specification to deal with quantifiable probabilities and presented a model for the logic applicable to DES.

The rest of the chapter is organized as follows: Section 4.1 presents the syntax of our logic, which is based on a Pnueli type of linear time temporal logic, Section 4.2 presents the semantics of the language and Section 4.3 an axiomatic proof system for it.

4.1 Syntax of the language

Temporal logic is a type of logic that uses an enriched set of symbols to describe time-related formulas as well as classical logic expressions. The language is based on a set of symbols used to construct *terms* and *formulas*.

Elementary set theory is the starting point of any mathematical logic. The development of set theory was motivated by several paradoxes that threatened the foundations of mathematics

[59]. Attempts to avoid the paradoxes lead to the development of the Zermelo's axiomatic system. Several mathematicians enriched the theory and the result is the so called *von Neumann - Bernays - Gödel (NBG) class theory.* This framework axiomatizes the concept of class. In the NBG theory there are two types of classes: those that satisfy Zermelo's axiomatic system, called *sets* and others, called *proper classes*, that don't satisfy it. In the sequel we consider this axiomatic of the set theory.

The logic that we consider is a multi sorted one [58] and therefore, the universe of the logic is indexed over a set of indexes (sorts) I. *Formulas* are build using classical logical connectives $(\wedge, \vee, \neg, \rightarrow)$ over a sorted set of variables, in the standard way. In addition basic temporal operators like $\Diamond$ (read as "eventually"; also read as "sometimes"), $\Box$ (read as "henceforth"), $\bigcirc$ (next) and $\mathcal{U}$ (until), are also considered.

The minimal alphabet that we need for the definition of our temporal logic can be described as follows:

Definition 4.1 *The **alphabet** of the logic consist of the following symbols:*

- *Logical connectives:* $\neg$, $-$ *and the existential quantifier* $\exists$.
- *For each sort* $i \in I$ *a set of variables:* $V_i = \{v_{i,0}, v_{i,1}, \ldots, v_{i,n}\}$.
- *Equality symbols (=) defined for some sorts* $i \in I$.
- *Parenthesis:* $), ($.
- *For each* $n \in \mathbb{N}$ *and each* $(n+1)$*-tuple* $< i_1, \ldots, i_n, i_{n+1} >$ *of sorts there is a set of functional symbols* $\{f_{i,0}, f_{i,1}, \ldots, f_{i,m}\}$.
- *For each* $n \in \mathbb{N}$ *and each* n*-tuple* $< i_1, \ldots, i_n >$ *of sorts there is a set of relational symbols* $\{p_{i,1}, p_{i,2}, \ldots, p_{i,m}\}$.
- *Temporal operators:* $\bigcirc, \Diamond, \mathcal{U}$.
- *Path operators:* $\|$ *(parallel) and* $|$ *(sequential).*

■

Some connectives are missing from the definition because they can be defined in terms of the already existing ones. Constants do not appear directly in the definition because we consider them

as function symbols with zero arity. Using the alphabet from Definition 4.1 terms can be built by applying a set of rules. The logic that we consider is a multi sorted one. Therefore, the logic is defined over several indexed domains D_i, each domain containing its variables, quantifiers and equality. The function symbols can be defined over several sorts $f : D_1 \times D_2 \times \ldots \times D_n \longrightarrow D_m$, like the relational symbols, defined as $r : D_1 \times D_2 \times \ldots \times D_n \longrightarrow \mathcal{B}$ (where $\mathcal{B}$ is the boolean domain of $\{T, F\}$, with T and F denoting the notions of *true* and *false* as defined by Tarski).

Usually the set of variables used in the definition of some temporal logic is partitioned into two categories [55, 70]: a set of variables that change with time, called either *local* or *flexible*, and a set formed by those that do not change with the passing of time. This latter variables are called *global* or *rigid*.

Definition 4.2 *A term[70] in the language is defined as following:*

- *All variable symbols of sort i are terms of sort i.*
- *If $t_1, \ldots, t_n$ are terms of sorts $< i_1, \ldots, i_n >$ and f is a function symbol of sort $< i_1, \ldots, i_n, i_{n+1} >$ then $f(t_0, \ldots, t_n)$ is a term of sort i_{n+1}.*
- *if t is a term $\bigcirc t$ is also a term.*
- *No other strings of symbols are terms.*

■

A restriction has to be imposed, as one can observe, on the formation of terms from function symbols, due to the multi sorted nature of the logic. Specifically, a term t has to have the same sort as the parameter it replaces.

From terms, which denote the objects of our universe, formulas can be built. Therefore, the next step in the definition of the logic is to present how formulas can be built. Formulas are an important part of any logic, since they make assertions about the composing terms.

Definition 4.3 *A **well-formed formula (wff)** [70] of temporal logic is defined inductively as follows:*

- *If t_1 and t_2 are terms of the same sort i then $t_1 = t_2$ is a formula of sort i.*

- *If $t_1, t_2, \ldots, t_n$ are terms of sorts $i_1, i_2, \ldots, i_n$ and p is an n-ary relation symbol (predicate) of sort $< i_1, i_2, \ldots, i_n >$ then $p(t_1, t_2, \ldots, t_n)$ is a formula.*
- *If w is a formula then $(\neg w)$, $(\bigcirc w)$, $\mid w$ and $(\Diamond w)$ are also formulas.*
- *If w_1 and w_2 are formulas then $(w_1 \rightarrow w_2)$, $(w_1 \parallel w_2)$ and $(w_1 \mathcal{U} w_2)$ are also formulas.*
- *If x is a variable and w is a formula then $\exists x : w$ is a formula.*
- *No other string of symbols is a formula.*

■

A formula formed by combining terms with the help of a predicate $p(t_1, t_2, \ldots, t_n)$ is called an *atomic formula*. Predicate symbols, due to their multi sorted format, can form new formulas only by substituting the parameters with terms of the proper sort. Equality is also affected, because both terms in an equality formula have to have the same sort. Therefore, a separate equality symbol has to be defined for different sorts of the language.

If a variable has at least one occurrence in a formula that is not within the scope of a quantifier that variable is called *free*. Let us now define a *substitution* procedure, that tell us how to substitute a term t for a variable x in a formula w at places where x occurs free [70].

Definition 4.4 *Let $g_1, g_2, \ldots, g_n$ be terms such that for all i, the variable x_i has the same sort as g_i. Then given a formula w a new formula $w \left[\begin{smallmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{smallmatrix}\right]$ can be obtained by applying the following rules:*

- *Terms:*

$$x \left[\begin{smallmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{smallmatrix}\right] \triangleq \begin{cases} x & \text{if } x \neq x_i \text{ for all } i \\ g_i & \text{if } x = x_i \end{cases}$$

$$f(t_1, \ldots, t_n) \left[\begin{smallmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{smallmatrix}\right] \triangleq f(t_1 \left[\begin{smallmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{smallmatrix}\right], \ldots, t_n \left[\begin{smallmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{smallmatrix}\right])$$

$$(\bigcirc w_1) \left[\begin{smallmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{smallmatrix}\right] \triangleq \bigcirc(w_1 \left[\begin{smallmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{smallmatrix}\right])$$

- *Formulas:*

$$(t_1 = t_2) \left[\begin{smallmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{smallmatrix}\right] \triangleq t_1 \left[\begin{smallmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{smallmatrix}\right] = t_2 \left[\begin{smallmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{smallmatrix}\right]$$

$$p(t_1, \ldots, t_n) \left[\begin{smallmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{smallmatrix}\right] \triangleq p(t_1 \left[\begin{smallmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{smallmatrix}\right], \ldots, t_n \left[\begin{smallmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{smallmatrix}\right])$$

$$(\neg w_1)\begin{bmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{bmatrix} \triangleq \neg(w_1 \begin{bmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{bmatrix})$$

$$(w_1 \rightarrow w_2)\begin{bmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{bmatrix} \triangleq (w_1 \begin{bmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{bmatrix} \rightarrow w_2 \begin{bmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{bmatrix})$$

$$(\Diamond w_1)\begin{bmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{bmatrix} \triangleq \Diamond(w_1 \begin{bmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{bmatrix})$$

$$(w_1 \mathcal{U} w_2)\begin{bmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{bmatrix} \triangleq (w_1 \begin{bmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{bmatrix} \mathcal{U} w_2 \begin{bmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{bmatrix})$$

$$(\forall v : w_1)\begin{bmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{bmatrix} \triangleq \begin{cases} (\forall v : w_1 \left[g_1 \cdots g_{i-1} {}^{x_1 \ldots x_{i-1} x_{i+1} \ldots x_n} g_{i+1} \cdots g_n\right]) & \text{if } x = v \text{ for some } i \\ (\forall v : w_1 \begin{bmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{bmatrix}) & \text{if } x \neq v \text{ for all } i \end{cases}$$

$$(w_1 \parallel w_2)\begin{bmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{bmatrix} \triangleq (w_1 \begin{bmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{bmatrix} \parallel w_2 \begin{bmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{bmatrix})$$

$$(| w)\begin{bmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{bmatrix} \triangleq| (w \begin{bmatrix} x_1 \ldots x_n \\ g_1 \ldots g_n \end{bmatrix})$$

■

4.2 Semantics of the language

Semantics of logics are represented over mathematical constructions on sets. For the multi sorted logics semantics are defined over *structures* [24], that are algebraic structures built on sets. Definition 4.5 formalizes the notion of structure, as it is usually used in logic. Let I be a set of sorts.

Definition 4.5 *A multi sorted* ***structure*** *[24] is a pair $S = (A, m)$ where A is a multi sorted set $A = (A_{i_1}, \ldots, A_{i_n})$ called the* ***universe*** *of S and m is a map defined on S satisfying the following properties:*

- *For every n-ary relation symbol R of sort $< i_1, \ldots, i_n >$, $m(R) \subseteq A_{i_1} \times \ldots \times A_{i_n}$.*
- *For every function symbol f of sort $< i_1, \ldots, i_n, i_{n+1} >$ $m(f)$ is a function $m(f) : A_{i_1} \times \ldots \times A_{i_n} \longrightarrow A_{i_{n+1}}$*

■

Constants do not appear in Definition 4.5, because they are considered as 0 arity functions. Therefore, presenting them in the definition specifically, is not necessary. For each constant of sort i, the structure S assigns an element of A_i. In the case of variables, the so called *assignment* gives the interpretation [59].

Definition 4.6 *An assignment of a structure $\mathcal{S}$ is a map which assigns a value in A_i for each variable of sort i.* ■

In a classical logic language [59] an interpretation is a pair $(\mathcal{S}, \beta)$ where $\mathcal{S}$ is a structure and β an assignment on that structure. Kripke invented the semantics of modal logic. It is defined over triples $(\mathcal{S}, \beta, \varsigma)$, formed by adding the notion of *trajectory* over the structure $\mathcal{S}$. Temporal logic adopts the same semantics description with the difference that the trajectory is considered to represent moments in time. A *trajectory* describes the dynamic behavior of the structure and regardless of the sort over which it is constructed it is related with the passing of time. Usually, the *trajectory* is built over a sort of *states.*

Definition 4.7 *An interpretation is a triple $(\mathcal{S}, \beta, \varsigma)$, where $\mathcal{S}$ is a structure, β is an assignment is $\mathcal{S}$ and ς is a trajectory on $\mathcal{S}$.* ■

If the logic is evaluated over a linear enumeration of time moments it is called *linear time* temporal logic. If, on the other hand, the logic is interpreted over a tree like structure of time the logic is called *branching time* temporal logic. The linear time view [55, 80, 21] presents the computations of a concurrent system as being interleaved. This is true for systems that create concurrency by time-sharing. However, it distorts the reality when we deal with a system capable of real parallelism, as it is the case of a distributed system. The *branching time temporal logic* [22, 21, 81] considers a branched view of time, where each branch represents a *possible* run of the system. It is important to notice that this view has a semantics over the universe of all the possible executions of the system. Both views are global, in the sense that they describe the system monolithically.

The definition of *structures and interpretations* gives us the relation with the framework of the previous chapter. In the definition of the DEA, from the previous chapter, the set of time values was considered to be R the set of real numbers. As previously discussed we consider a binary relation over the set $<$. In the domain of logic this relation is a predicate [24, 19].

Let us consider the semantical structures to be DEAs (the identification is immediate) together with the relational symbol $<$, defined over the set of time values. A *trajectory* over a DEA is a sequence of states over the state set of the DEA. Thus, an interpretation in the DEA framework is a triple $((M, \mathcal{B}, <), \beta, \varsigma)$, where $M = (T, Q, E, Y, \delta, \Gamma, q_0)$ is a DEA, $\mathcal{B}$ is the universe of boolean values, β is an assignment and ς is a trajectory over $Q \times T$. The sets T, Q, E, Y and $\mathcal{B}$ are the

sets of our universe, (δ, Γ, q_0) the functional symbols and $<$ is a predicate over the time values. Thus, $(M, \mathcal{B}, <)$ by Definition 4.5 is a relational structure and it can be used as such.

A distributed system $\mathcal{DS}$ is a double $(\mathcal{P},\mathcal{C})$ composed of a set of sequential processes $\mathcal{P}$ and a set of communication channels $\mathcal{C}$. Since sequential processes are also made up of DEAs, we can consider the system as being composed of a single set DEAs and several operations that allow composition. Let us now consider only the composing set of DEAs without any structure. In this case the state space of $\mathcal{DS}$ is the Cartesian product of the state spaces of the components. The desired behavior of the distributed system is usually a subset of the Cartesian product of the components, that can be obtained by imposing restrictions over the system. This restrictions are *relations* between states and events of the composing DEAs. By adding this set of relations to the description of a distributed system we have a relational structure over which we can interpret the semantics of our logics. In our relational structure $\mathcal{DS}t$ the sets of the individual DEAs are considered to be distinct. Each DEA is considered with is own trace ς_i. Thus, the interpretation contains a *set of trajectories* over which we consider our semantics.

As a preliminary step in the definition of *satisfaction* we associate with every interpretation $\mathcal{I} = \{\mathcal{S}, \beta, \varsigma\}$ and every term t an element $\mathcal{I}(t)$ using induction on terms [70]:

Definition 4.8

- *For each variable* v, $\mathcal{I}[v] = A(v)$.
- *For each* n*-ary function symbol* f *and terms* $t_1, t_2, \ldots, t_n$ *(all of appropriate sorts)*

$$\mathcal{I}[f(t_1, \ldots, t_n)] = \mathcal{I}[f](\mathcal{I}[t_1], \ldots, \mathcal{I}[t_n])$$

.

- *For each* $(n+1)$*-ary predicate symbol* P *and terms* $t_1, t_2, \ldots, t_n$ *(all of appropriate sorts)*

$$\mathcal{I}[P(t_1, \ldots, t_n)] = \mathcal{I}[P](\mathcal{I}[t_1], \ldots, \mathcal{I}[t_n])$$

.

- *For any term* t, $\mathcal{I}[\bigcirc t] = \mathcal{I}^{(1)}[t]$. *If* $\mathcal{I}$ *is an interpretation with* $\varsigma = s_0 s_1 s_2 \ldots$ *then* $\mathcal{I}^{(1)}$ *denotes the interpretation with* $\varsigma = s_1 s_2 \ldots$.

■

Temporal logics with $\bigcirc$ (*next*) are not invariant under shuttering [45, 46]. This means that, if we consider a state sequence for a system $s_0 s_1 s_2 \ldots s_n \ldots$ and another one $s_0 s_2 s_4 \ldots$ that generates the same behavior as the first one it will take us to two different temporal logic description if we use the $\bigcirc$ operator. A solution would be to use different variables to denote the beginning and the end of change, and thus, to leave the time interval over which the change takes place unspecified. This has the drawback, in my opinion, that it complicates matters too much by making distinction between the "left hand side" and the "right hand side" of a variable. Despite the specified drawback we have chosen to use the *next* operator in the language, due to the natural descriptions that it generates when used to describe discrete event systems.

The *satisfaction relation* is considered over a *distributed system structure* $\mathcal{S} = (\mathcal{DS}, \mathcal{B}, \mathcal{R})$ where $\mathcal{DS}$ is a distributed system, $\mathcal{B}$ is the set of true and false values, $\mathcal{R}$ is a set of relations over elements of $\mathcal{DS}$. As discussed above in this case ς is a *set* of trajectories.In the following definition we utilize the notation $M, x \models w$ to mean that formula w is true in the model M over he trajectory (trajectories) x.

Definition 4.9 ***Satisfaction Relation.*** *A formula w is said to be satisfied by an interpretation $\mathcal{I} = \{M, \varsigma\}$, where $M = \{S, \beta\}$ is a structure, denoted by $M \models w$ if:*

- $M \models (t_1 = t_2)$ *iff* $\mathcal{I}[t_1] = \mathcal{I}[t_2]$ *(where t_1 and t_2 have the same sort).*
- $M \models P(t_1, t_2, \ldots, t_n)$ *iff* $(\mathcal{I}[t_1], \ldots, \mathcal{I}[t_n]) \in \mathcal{I}[P]$.
- $M \models (\neg w)$ *iff it is not the case that* $M, s \models w$.
- $M \models (v \rightarrow w)$ *iff either* $M, s \models v$ *or it is not the case that* $M, s \models w$.
- $M, x \models (\bigcirc w)$ *iff* $M^{(1)} \models w$.
- $M, x \models (\Box w)$ *iff for all* $i \geq 0$, $\models_{M^{(i)}} w$.
- $M, x \models (v \mathcal{U} w)$ *iff there exists an* $n \geq 0$ *such that* $M^{(n)} \models w$ *and for all* k, $0 \leq k < n$, $M^{(k)} \models v$.
- $M, T \models (\mid w)$ *iff there exists* $x \in T$ *such that* $M, x \models w$.
- $M, \{x_1, x_2\} \models (w_1 \parallel w_2)$ *iff* $M, x_1 \models w_1$ *and* $M, x_2 \models w_2$ *or vice versa.*

■

For the satisfaction relation of the parallel ($\|$) and serial ($|$) operators we need to consider a *set* of structures, each of which has a corresponding *trajectory*.

If M *satisfies* [19] a formula w, this is the same as saying that M is a *model* of w.

Definition 4.10 *An interpretation $\mathcal{I}$ is a* ***model*** *for a set of formulas Φ, denoted by $\mathcal{I} \models \Phi$ if, for all $\varphi \in \Phi$ we have $M \models \varphi$.* ■

Definition 4.11 *We say that Φ_1 is a consequence of Φ_2 (written $\Phi_2 \models \Phi_1$) iff every interpretation which is a model of Φ_2 is also a model of Φ_1.* ■

Using the notion of consequence we are now able to define the notions of *validity, satisfiability and logical consequence.*

Definition 4.12

- *A formula w is* ***valid*** *(written $\models w$) iff $\phi \models w$.*
- *A formula w is* ***satisfiable*** *(written* ***Sat*** *w) iff there exists an interpretation which is a model of w. A set of formulas Φ is satisfiable (written* ***Sat*** *Φ) iff there is an interpretation which is a model of all formulas in Φ.*
- *Two formulas w_1 and w_2 are* ***logically equivalent*** *iff $w_1 \models w_2$ and $w_2 \models w_1$.*

■

The expression $\phi \models w$ tells us that a formula w is valid if it is satisfied by any model. It is sometimes useful to restrict the validity of a formula to a class of models $\mathcal{C}$ (in our case to the class of DEAs and distributes systems models). A formula w which is true for every model in $\mathcal{C}$ is said to be $\mathcal{C}$-valid, denoted by $\mathcal{C} \models w$.

Let us consider Definition 4.4. One question that comes into ones mind is how does the interpretation hold over the substitution of terms. The following lemma [19] gives us the answer.

Lemma 4.1 ***Substitution Lemma:***

- *For every term t, $\mathcal{I}\left(t\left[\begin{smallmatrix} t_0 \ldots t_n \\ x_0 \ldots x_n \end{smallmatrix}\right]\right) = \mathcal{I}\left[\begin{smallmatrix} \mathcal{I}(t_0) \ldots \mathcal{I}(t_n) \\ x_0 \ldots x_n \end{smallmatrix}\right](t)$.*
- *For every formula φ, $\mathcal{I} \models \varphi\left[\begin{smallmatrix} t_0 \ldots t_n \\ x_0 \ldots x_n \end{smallmatrix}\right]$ iff $\mathcal{I}\left[\begin{smallmatrix} \mathcal{I}(t_0) \ldots \mathcal{I}(t_n) \\ x_0 \ldots x_n \end{smallmatrix}\right] \models \varphi$.*

■

Proof: By induction on terms and formulas using Definition 4.4.

■

As previously said some of the well known logical connectives can be defined in terms of others. Here are some that are most likely to be used:

Name	Symbol	Abbreviates
disjunction	$(w_1 \vee w_2)$	$((\neg w_1) \rightarrow w_2)$
conjunction	$(w_1 \wedge w_2)$	$\neg(w_1 \rightarrow (\neg w_2))$
biconditional	$w_1 \leftrightarrow w_2$	$(w_1 \rightarrow w_2) \wedge (w_2 \rightarrow w_1)$
henceforth	$(\Box w)$	$(\neg(\Diamond(\neg w)))$
precedes	$w_1 \mathcal{P} w_2$	$(\neg((\neg w_1)\mathcal{U} w_2))$

Table 4.1: Frequently used connectives and their equivalences

The appearance of formulas can be simplified by applying rules that add precedence. This way the number of parenthesis can be (hopefully) reduced.

- The unary operators apply to the expression that follows them.
- The $\neg$ operator has the greatest precedence, followed by $|, \Box, \Diamond, \mathcal{U}, \|\rightarrow$.

4.3 The deductive system

To obtain valid formulas, from ones that we already know as valid, we employ the notion of *deductive system*. Such a system consists of two elements:

- A set of *axioms*. This is a set of formulas that are considered to be true.
- A set of *deduction rules*. They provide the pattern that allow us to derive new formulas from old ones.

There are many ways in which we can consider a deductive system. In the sequel we will consider a system with an *infinite* number of axioms [70] and a set of deduction rules. The original deductive system of Manna and Pnueli [54] contained only a finite set. From the initial set of rules several others can be derived and used. Some the deduction rules are called *theorems*

as is the case with formulas that are derived from a set of axioms. A better term, suggested by Enderton [24], is that of *metatheorems*, but since the confusion does not seem likely to arise, we will use the word theorem as in [70].

4.3.1 Axioms of the system

As in any proof system some of the valid formulas are considered as axioms and the others are considered to be deduced using inference rules. The original proof system [54] is divided in three parts: the *general, domain and program* parts, each with their corresponding axioms. The *general part* is used to prove temporal properties that hold on arbitrary interpretations. Thus, this part will be maintained in our axiomatic system. The *domain and program* layers will be changed to suite our needs. The axioms 1 to 8 are from [70].

Axiom 1 *w, where w is an instance of a propositional tautology;*

Axiom 2 $\neg \bigcirc w \leftrightarrow \bigcirc \neg w$;

Axiom 3 $\Box(w_1 \rightarrow w_2) \rightarrow (\Box w_1 \rightarrow \Box w_2)$

Axiom 4 $w_1 \mathcal{U} w_2 \rightarrow \Diamond w_2$

Axiom 5 $\bigcirc(w_1 \rightarrow w_2) \rightarrow (\bigcirc w_1 \rightarrow \bigcirc w_2)$;

Axiom 6 $\Box w \rightarrow w \wedge \bigcirc \Box w \wedge \bigcirc w$

Axiom 7 $w_1 \mathcal{U} w_2 \leftrightarrow [w_2 \vee (w_1 \wedge \bigcirc(w_1 \mathcal{U} w_2))]$

Axiom 8 $\Box(w \rightarrow \bigcirc w) \rightarrow (w \rightarrow \Box w)$

The proposed logic also has the equality symbol. In order to be able to use it in deductions several axioms have to be added.

Axiom 9 *$t = t$, for any term t of sort i for each the equality symbol is defined.*

Axiom 10 *$(t_1 = t_2) \rightarrow (w(t_1, t_1) \rightarrow w(t_1, t_2))$, for any terms t_1 and t_2 and any formula $w(t_1, t_1)$ and $w(t_1, t_2)$, where $w(t_1, t_2)$ is obtained from $w(t_1, t_1)$ by replacing with t_2 some of the occurrences of t_1 that are not within the scope of any temporal operator.*

The following two axioms were added by Ostroff [70] in order to extend propositional temporal logic to predicate temporal logic.

Axiom 11 *For any function f of sort $< i_1, i_2, \ldots, i_n, i_{n+1} >$ and terms $t_1, t_2, \ldots, t_n$ of corresponding sorts* $\bigcirc f(t_1, t_2, \ldots, t_n) = f((\bigcirc t_1), (\bigcirc t_2), \ldots, (\bigcirc t_n))$.

Axiom 12 *For any $n-ary$ predicate letter P of sort $< i_1, i_2, \ldots, i_n >$ and terms $t_1, t_2, \ldots, t_n$ of corresponding sorts*

$$\bigcirc P(t_1, t_2, \ldots, t_n) \rightarrow P((\bigcirc t_1)(\bigcirc t_2), \ldots, (\bigcirc t_n))$$

The second part of the original proof system of Manna and Pnueli [54] is called the *domain part*. In this part axioms are added if we intend to manipulate special domains. For example if manipulation of natural numbers is desired, the Peano's set of axioms is needed. For this part of the deduction system we will add a set of axioms regarding the *sequential* and *parallel* operators.

Axiom 13 $|\,(w_1 \rightarrow w_2) \rightarrow (|\, w_1 \rightarrow |\, w_2)$

Axiom 14 $\|_{i=1}^{k}\, w_i \rightarrow \wedge_{i=1}^{k}(|\, w_i)$

Axiom 15 $|\, w \rightarrow w$

Axiom 16 $|\, \bigcirc w \rightarrow \bigcirc\, |\, w$

Axiom 17 $\Box\, \|_{i=1}^{k}\, (\bigcirc^{(i)} |\, w_i) \rightarrow \Box(\wedge_{i=1}^{k}(\bigcirc^{(i)} |\, w_i))$

The introduction of the *sequential* (|) and *parallel* (‖) operators allows us to represent the parallelism without interleaving. This is basically what happens in a truly parallel or distributed system.

The set of axioms for the program part in our proof system is empty.

4.3.2 Rules of inference

In the sequel the notation $\frac{w_1, w_2, \ldots, w_n}{w_{n+1}}$ will mean that one may infer the formula w_{n+1} from the formulas $w_1, w_2, \ldots, w_n$. Also the notation $\vdash w$ means that the formula w is a theorem of our proof system, and $\Phi \vdash w$ means that w can be deduced from the set Φ of formulas. Our deductive system uses only two rules. From this two others can be inferred and subsequently used.

Modus Ponens: For any formulas w_1 and w_2

$$\frac{w_1,\ w_1 \rightarrow w_2}{w_2}$$

. ■

Generalization rule: If $\{w_1, w_2, \ldots, w_n\} \vdash w_{n+1}$, then

$$\{\Box w_1, \Box w_2, \ldots, \Box w_n\} \vdash \Box w_{n+1}$$

. ■

From the above deduction rules several metatheorems can be obtained. Here are some of them, as described in [37].

Propositional reasoning(PR): If

$$\wedge_{i=1}^{n} w_i \rightarrow w_{n+1}$$

is an instance of a tautology, then

$$\frac{w_1, w_2, \ldots, w_n}{w_{n+1}}$$

■

Deduction theorem: $\Phi \cup \{w_1\} \vdash w_2$ if and only if $\Phi \vdash w_1 \rightarrow w_2$ for any set of formulas $\Phi \cup \{w_1, w_2\}$. ■

Derived computational induction rule:

$$\frac{\Box(w_1 \rightarrow (w_2 \wedge \bigcirc w_1))}{\Box(w_1 \rightarrow \Box w_2)}$$

■

Right until introduction:

$$\frac{\Box(w_1 \wedge w_3 \rightarrow \bigcirc w_2)}{\begin{array}{c} w_1 \rightarrow \Diamond w_3 \\ \Box(w_1 \rightarrow (\bigcirc w_2 \vee \bigcirc w_1)) \end{array}}$$
$$w_1 \rightarrow (w_1 \mathcal{U} w_2)$$

■

Right precedes introduction:

$$\frac{\Box(w_1 \rightarrow \neg w_3 \wedge (w_2 \vee \bigcirc w_1))}{\Box(w_1 \rightarrow w_2 \mathcal{P} w_3)}$$

■

***$\mathcal{P}$-chain*:**

$$\frac{\Box\{v_i \rightarrow \neg w \wedge [\vee_{j<i} v_j \vee \bigcirc(\vee_{k=1}^{i} v_k)]\} \quad \textit{for all } i, \ 0 < i \leq n}{\Box\{\vee_{i=1}^{n} v_i \rightarrow v_0 \mathcal{P} w\}}$$

■

Frame Theorem:

$$\vdash \Box[\bigcirc w(y_1, y_2, \ldots, y_n) \rightarrow w((\bigcirc y_1), (\bigcirc y_2), \ldots, (\bigcirc y_n))]$$

for any formula $w(y_1, y_2, \ldots, y_n)$ where y_1, y_2, $\cdots$, y_n are the only local variable symbols in $w(y_1, y_2, \ldots, y_n)$ and formula $w((\bigcirc y_1), (\bigcirc y_2), \ldots, (\bigcirc y_n))$ is obtained by replacing each of these local variable symbols y_i in w by $(\bigcirc y_i)$. ■

Mathematical reasoning: If the formula $(\wedge_{i=1}^{n} w_i) \rightarrow w_{n+1}$ is a domain axiom, then

$$\frac{w_1, w_2, \ldots, w_n}{w_{n+1}}$$

■

4.3.3 Soundness and Completeness

In this section, we try to establish the *soundness* and *completeness* of our deductive calculus. As mentioned, $\vdash w$ means that w is a theorem of our deductive system and $\Gamma \vdash w$ means that w can be deduced from a set of formulas Γ. Also, the notation $\Gamma \models w$ means that Γ semantically implies w. More precisely a set of formulas Γ *semantically implies* another formula w if for any model M of Γ is also a model for w.

The main result of the section is that our deductive system is complet. This is translated into the following theorem:

Theorem 4.1 *For all formulas w and sets of formulas Γ:*

$$\Gamma \models w \textit{ iff } \Gamma \vdash w$$

. ■

There is a little bit of confusion on the naming of the theorem, mainly because when Kurt Gödel first presented the theorem, it had a bidirectional implication (iff). Henkin presented another method of proving the theorem, and separated theorem into two distinct parts: one for soundness ($\Gamma \vdash w \rightarrow \Gamma \models w$) and one for the reverse implication which he called completeness ($\Gamma \models w \rightarrow \Gamma \vdash w$).

Depending on Γ there are two types of soundness as presented in the following definition.

Definition 4.13 *A proof system is:*

- *sound iff* $\vdash w$ *implies* $\models w$ *for all* w.
- *strongly sound iff* $\Phi \vdash w$ *implies* $\Phi \models w$ *for any* Φ *and* w.

■

Theorem 4.2 ***(Soundness:)*** *For all formulas* w *and sets of formulas* Γ *if* $\Gamma \vdash w$ *then* $\Gamma \models w$ ■

It is also worth mentioning that there are several forms of completeness for logical systems. The one we are interested in is called *deductive completeness* and can be specified by the theorem:

Theorem 4.3 ***(Completeness:)*** *For all formulas* w *and sets of formulas* Γ *if* $\Gamma \models w$ *then* $\Gamma \vdash w$. ■

The proof for completeness of our system follows the *Henkin-Hansenjäger* method [42, 43]. The method has several steps that have to be followed. First let us define the following notion of consistency.

Definition 4.14 *A finite set of formulas* Γ *is called consistent if for no finite number of formulas* $w_1, \ldots, w_n$ *from* Γ, $\vdash \neg(w_1 \wedge \ldots \wedge w_n)$ ■

Using the above definition, the following lemmas can be proved:

Lemma 4.2 *Let* Γ *be a finite, consistent set of formulas. There is a finite, consistent and complete set* Γ^* *of formulas with* $\Gamma^* \subset \Gamma$. ■

From the above lemma the following result, called *Lindebaum's lemma* can be proved.

Lemma 4.3 *Each consistent set of formulas* Γ *is contained in some maximally consistent one.* ■

A maximally consistent set Γ has the interesting property that,

- $\neg w \in \Gamma$ iff $w \notin \Gamma$,
- $w_1 \rightarrow w_2 \in \Gamma$ iff $w_1 \in \Gamma \rightarrow w_2 \in \Gamma$.

Next step in the proof is the *truth lemma.*

Lemma 4.4 *For all maximally consistent sets* Γ *and all formulas* w, $M, \Gamma \models w$ *iff* $w \in \Gamma$. ■

Using the above lemmas, Theorem 4.3 can be proved. It is interesting to notice that, if we add Peano's axioms to the *domain layer* of the proof system, then the deductive system becomes incomplete [72]. This is due to the well known problem of incompleteness of theories built over natural numbers. In our logic, we avoid using Peano's axioms by not considering the time as a clock variable that has to be incremented [70, 37] but, as a normal variable of sort t. The set of time values, though, has defined over it two relations (<,=) that can be subsequently used in the logical language.

Chapter 5

A Software Framework for Specifying Distributed Systems

This chapter presents the syntax and semantics of a specification language. The language is based on LALR(1) [3] grammar that makes it easily implementable using an automatic parser generator like *yacc* or *bison*. Section 5.1 presents the syntax of the language. To make the task of specification more manageable the language is modular. The modularization helps the designer not only to verify parts of the design separately but also to reuse parts of a specification. Section 5.2 presents the semantics of the language using the model developed in Chapter 3 and the logical language developed in Chapter 4.

5.1 Syntax of the Language

The specification language described in this chapter was designed with several goals in mind:

- To support modularization. This was considered an important requirement, because by making the language modular, the task of verification of the design can be made easier and the specifications tend to be more readable.
- To use the logical and temporal operators described in the previous chapter. We wanted that the specification language to reflect the theoretical framework described in the previous chapters.

- To be simple to use. This is a general requirement for any programming language, since by complicating things the users can become alienated and stop using the language.

Following the above criteria, a language called *Temporal Logic Language (TLL)*, which is an improved version of the language described in [44], resulted.

The syntax of a programming language can be described in several ways. The most common type of syntactic presentation is called *concrete syntax* description [3], which treats the language as a set of strings *strings*. Concrete syntax is described by a set of rules called *productions*. Depending on the complexity of the productions the class of all languages can be divided into several categories. Practical programming languages are generated by at least a *context free* grammar. Our the set of productions is designed to be LALR(1). This design decision was made because of the limitations that the automatic parser generators, available to us, have. For more complicated types of syntaxes the parser would have had to be written by hand, which is extremely difficult and time consuming.

Another type of syntax description that is extensively used is the *abstract syntax* [64]. This type of description treats the language as a set of *trees*. The important advantage of this method over the previous one is that trees have an *unambiguous* structure: there is only one way to construct a tree from its immediate subtrees.

The simplest thing that can be done in almost any programming language is to write comments. Although not necessary for the description of a language, comments improve program readability and thus, were considered useful. In *TLL* they start with the character @. The rest of the line, after the comment character, is omitted. Comments do not appear in the grammar since they are eliminated by the lexical analyzer.

The grammar of *TLL* will be given using the *Backus-Naur form* (*BNF*). All *BNF* descriptions in this thesis are bound to the following rules:

- All nonterminal symbols used in the grammar are written in small letters (for example *program*).

- All terminal symbols are written in capitals (for example *MODULE*).

A valid program in *TLL* is composed of several entities: module declarations, global constant declarations and the system description. The main structure available in the language is the **Module**. Any number of module declarations are allowed in the language. Global constants

can also be declared in the program. The declarations of modules and constants can alternate. After the **Module** and **Constant** declarations comes the description of the **System**. Table 5.1 presents the *BNF* grammar for the constant and module declarations.

program	:: ext_def_list system_declaration
ext_def_list	:: ext_def_list ext_def
	| ε
ext_def	:: constant_declaration
	| module_declaration

Table 5.1: *BNF* description of a *TLL* program

Before we continue with the presentation of the syntax, let us give an example of a *TLL* program. The modules content and the system description, in Example 5.1, are empty. In *TLL* the scope of the identifiers is within the unit were they were declared. Therefore, all the global constant declaration and module names have to be different, otherwise a compiling error is generated. All global constant declarations are visible in the modules and the system declaration. The names declared *inside* a module or *inside* the system specification are not visible outside. All such declarations take precedence over the global declarations.

Example 5.1 @ Example of a valid TLL program

Constants *one = 100, two = 2, three ,forty[40], fifty[one];*

Module *M1* {}

Module *M2* {}

Constant *last;*

Module *M3* {}

System {}

■

Constants are declared as lists of symbols terminated by semi column (Example 5.1). Each constant has been assigned a numerical value. This can be done explicitly (see constants *one*, *two*

in Example 5.1), or implicitly (constants *three, forty, fifty* and *last* in Example 5.1). Constants can also be indexed, by declaring an *array* of constants (Ex. *forty[40]*). The *BNF* grammar for constant description is presented in Table 5.2.

constant_declaration	:: CONSTANT constant_list SEMI
constant_list	:: constant
	| constant_list COMA constant
	| ε
constant	:: NAME simple_or_array constant_value
simple_or_array	:: simple_or_array LP declared_value RP
	| ε
declared_value	:: NAME
	| INTEGER
constant_value	:: EQUOP fnumber
	| ε
fnumber	:: INTEGER
	| FLOAT

Table 5.2: *BNF* description of constant declarations

Module descriptions start with the keyword **Module.** The keyword is followed by the *name* of the module and an optional *input - output description* (IOD). An IOD states the number of input and output lines that a module has and names them. By default the module has one input line and one output line respectively (Example 5.2). The input line description starts with the keyword **LIn** (LIne In) followed by a list of names that denotes the lines, while the output description starts with the keyword **LOut** (Line Out) followed by the names of the output lines.

The sets of events and outputs are global in the program. Each input and output line has a separate set of events. By default the sets are considered empty. Hence, if there is not any event (or output) declaration for a certain line (see line c in Example 5.2) the corresponding set is empty. Several lines of the same type can be described simultaneously by using an array declaration (*d[3]* in Example 5.2). The corresponding set of events (or outputs) is, in this case, described only for one of the lines.

Example 5.2 @ IO description of a module

Module *One* **[LIn**(*a,b,c,d[3]*);**LOut**(*out*)]

Events *a(e1,e2,e3);*

Outputs *out(o1,o2,o3);*

Events *b(e4(2.3,4.5),e6(3.1,5));*

Events *d(e10,e11);*

{}

■

Events are described as a list of symbols separated by comma and ending with semi column. They can be *timed* or *untimed*. Untimed events are described by their names only, as where their timed counterparts also have a description of the lower and upper time limits within which the event is active. Each time limit can be accessed by qualifying the name of the timed event with **TMin** for the lower time limit, and **TMax** for the upper time limit (Ex. $e4.TMin, e6.TMax$). Table 5.3 gives the grammar for the I/O description.

The module description is delimited by brackets. Inside the brackets, we can identify several constructions that are used to describe the module:

- A description of the state set and memory zone.
- A description of the output specifications.
- Specification of the transition time.
- Specification of the dynamics of the module.

Example 5.3 @ The specification of a module

Module *Example*

module_declarations	:: MODULE NAME io_lines io_description_list LC module_content RC
	| ε
io_lines	:: LP i_lines SEMI o_lines RP
	| ε
i_lines	:: LIN LB list_of_names RB
	| ε
o_lines	:: LOUT LB list_of_names RB
	| ε
list_of_names	name
	| list_of_names COMA name
io_description_list	:: io_description_list io_description
	| ε
io_description	EVENT evar_list SEMI
	| OUTPUT over_list SEMI
evar_list	:: events_for_one_line
	| evar_list COMA events_for_one_line
events_for_one_line	io_name LB list_of_events RB
io_name	:: NAME
	| ε
list_of_events	:: NAME simple_or_array timed_or_untimed
timed_or_untimed	:: LB fnumber COMA fnumber
	| ε
ovar_list	:: outputs_for_one_line
	| ovar_list COMA outputs_for_one_line
outputs_for_one_line	io_name LB list_of_names RB

Table 5.3: Grammar for module specification (1)

Event *(e0, e1(0,5), e2);*

{

State **s0, s1, s2;*

TTime = *1;*

Equ $s0 \wedge e0 \rightarrow s1$;

Equ $s1 \wedge e1 \rightarrow s2$;

Equ $s2 \wedge e2 \rightarrow s0$;

}

■

The state description is just an enumeration of states, which starts with the keyword **State**. Each state is separated by comma from the others and the declaration ends with semi column. An asterisk symbol (*) marks the initial state. The relation between the outputs and the states of the system is given by the *output specifications* (**Outspec**). An **Outspec** is a list of relations between a state and is corresponding output value. Each list starts with the keyword **Outspec** and ends with semi column.

Example 5.4 **Outspec** *s1 – > o1, s2 – > o2;*

■

Modules can also contain memory declarations. Such a declaration starts with the keyword **Memory** followed by a list of names, simple or arrays, that denote the memory locations. All declared memory variables can be subsequently used in the temporal logic equations. The initial state of the module is described by the initialization declaration (**Init**).

The value of the transition time of the module is given by assigning it to the predefined variable **TTime** (Transition Time). This variable has the default value 1. Each module has a time variable that keeps track of the local time. The variable can be accessed by declaring **TVariable** variable. More than one declaration is allowed but all the variables will indicate the same value.

The dynamics of the module is described by logical formulas. Each formula starts with the keyword **Equ** followed by a temporal logic formula. The *BNF* description for the description of modules is given in Tables 5.4 and 5.5.

module_content	:: module_content module_specs
	| ε
modules_specs	:: STATE svar_list SEMI
	| OUTSPEC list_of_out_specs SEMI
	| MEMORY list_of_names SEMI
	| CONSTANT constant_list SEMI
	| INIT init_list SEMI
	| EQUATION equation SEMI
	| TTIME EQUOP float_number SEMI
	| TVARIABLE NAME SEMI
	| INTDECL int_list SEMI
list_of_out_specs	:: out_spec
	| list_of_out_specs COMA out_spec
	| ε
out_spec	:: LB out_spec RB
	| NAME OUTVAL NAME
	| forall_list out_spec1
forall_list	:: FORALL list_of_vars IN domain
list_of_vars	:: NAME
	| list_of_vars COMA NAME
out_spec1	:: out_spec2 OUTVAL out_spec2
out_spec2	:: LB out_spec2 RB
	| NAME simple_or_array out_spec3
out_spec3	:: EQUOP NAME simple_or_array
	| ε
domain	:: LP declared_value DOTDOT declared_value LP
	| FLOAT

Table 5.4: Grammar for module specification (2)

init_list	:: init_spec
	| init_list COMA init_spec
	| ε
init_spec	:: NAME EQUOP NAME
	| LB init_spec RB
	| forall_list init_spec1
init_spec1	:: LB init_spec1 RB
	| NAME simple_or_array EQUOP NAME simple_or_array
svar_list	:: state_variable
	| svar_list COMA state_variable
	| ε
state_variable	:: state_name simple_or_array
	| INITSTATE state_name
state_name	:: NAME
float_number	:: INTEGER
	| FLOAT

Table 5.5: Grammar for module specification (3)

Module and constant declarations are followed by the description of the **System**. The system description has to end a *TLL* program and is composed of several parts.

- A *declaration part* that specifies all the *objects* that are running in the system. The object declarations can also be classified into three categories:
  - A list of declarations for the objects that compose the *plant*, as discussed in Chapter 3. The description of the plant starts with the keyword *Plant* (Example 5.5) followed by the object declarations.
  - A list of declarations for the objects that compose the *controller* part of the system. The parts that make the controller are declared after the keyword *Controller*.
  - A list of objects that describe the communication channels. This list of declarations starts with the keyword *Channel*.
- A *restriction part* that specifies all the restrictions imposed on the set of set of objects.
- A *connection* description part. This describes how the objects that make the system are interconnected.
- A *verification part* that accepts temporal logic expressions that are to be verified against the system.

Example 5.5 **System** {
Plant: **Client1** *p[3];*
Controller: **Client2** *c2;*

Restrict:
Generic *Client1(x,y),Client2(z);*
Equ **HF***(x.s2 ∧ (y.s1 ∨ y.s2) ∧ (z.s0)) ;*

Verify:
Equ **EV***(p[1].s2 ∧ p[2].s2 ∨ p[3].s2 ∨ c2.s2);*
}
■

system_description	:: SYSTEM LC system_content RC
	| ε
system_content	:: system_content system_specs
	| ε
system_specs	:: component
	| restrict_decl
	| verification_decl
component	:: PLANT COLUMN part_list
	| CONTROLLER COLUMN part_list
	| CHANNEL COLUMN part_list
	| CONNECTION COLUMN connection_list

Table 5.6: *BNF* description of the system (1)

All object declarations start with the name of a *module* followed by a list of symbols denoting the objects. The symbols are separated by *comma* and each list ends with *semi column*. There are two ways in which the objects can be declared: either as a list of individual symbols, each symbol denoting the name of the object (*c2* in the above example) , or as a set of indexed objects (*p[3]* in Example 5.5). All the declared *objects* are considered to be running in parallel. The maximum number of objects that can be declared is implementation dependent.

Another part of of the system description is represented by the list of connections. The objects that are part of the system can be interconnected with each other, by specifying links between their inputs and outputs. The connections description start with the keyword **Connection** followed by a list of connection descriptions. Each of the descriptions starts with the keyword **Equ** followed by the respective equation. The *BNF* grammar for connections description is given in Table 5.7.

connection_list	:: connection_list connection
	| ε
connection	:: EQUATION connect_equ SEMI
connect_equ	:: LB connect_equ RB
	| connect_equ1
	| forall_list connect_equ1
connect_equ1	:: LIN LB NAME simple_or_array LB
	NAME simple_or_array RB RB EQUOP
	LOUT LB NAME simple_or_array LB
	NAME simple_or_array RB RB

Table 5.7: *BNF* description for the connections

The *restrictions* are described in two subparts: one that declares *generic variables* and another that describes the restriction equations.

To implement the notion of bound variable, over sets of objects, we introduced in *TLL* the notion of *generic variable*. A generic variable takes any value from the set of objects those type is an instance of. To be more precise let us take a look at Example 5.5. The *generic* variable declaration starts with the keyword **Generic**. Each set of generic variables (x, y) is bound to a certain domain (Client1) by declaring them to take values over a certain set of objects. The domain is defined by the objects of a certain type (Client1) that exist in the system (*p[1],p[2],p[3]*).

The desired behavior of the system is, usually, only a subset of the Cartesian product of the state spaces of the composing objects. Parts of the state space are eliminated by imposing *restrictions* on the uncontrolled behavior of the system. This is consistent with the systemic model discussed in Section 3.7. Restrictions are described by temporal logic expressions. The terms of the expressions can be parts of the objects such as *states* or *events*. Any number of restrictions can be described. Table 5.8 presents the grammar of the restriction definitions.

The last part in the system specification is the *verification part*. It starts with the keyword **Verify**, followed by a list of properties that one wants to verify against the system. The grammar of the formulas accpeted by *TLL* is given in Table 5.9 and operator precedence is given in Table 5.10. The operators that are in the same line in the table have the same precedence. Those that are nearer the top of the table have higher precedence that the others. The compiler should also verify the *type* of the expressions, since the ¡ and ¿ operators are defined only for the sort of time.

The definitions of the terminal symbols is given in Tables 5.11 and 5.12. The set of terminal symbols is fixed and cannot be redefined. As one can see in Table 5.11 there are several forms for some of the terminals. This was chosen because the programs looked more readable, if both a singular and a plural form were included for some keywords. Also, the probability of syntactic errors due to misspelling decreases. The terminal symbol *NAME* stands for any string that start with a letter and it is not in the set of keywords. The terminal *INTEGER stands for any string formed only of numbers, while the terminal* FLOAT stands for a string composed of numbers that also contain the period character (Ex. 1.22). IEEE floating point description of reals is not accepted in the language.

Other symbols used in the language that appear in the *BNF* descriptions are given in Table 5.12

part_list	:: part_list part
	| ε
part	:: NAME system_part_list SEMI
system_part_list	:: new_part_name
	| system_part_list COMA new_part_name
new_part_name	:: NAME simple_or_array
restrict_decl	:: RESTRICT COLUMN generic_var_list restrict_equ_list
	| ε
generic_var_list	:: generic_var_list gdecl
	| ε
gdecl	:: GENERIC gvar_list SEMI
gvar_list	:: gvar
	| gvar_list COMA gvar
gvar	:: NAME LB l_var RB
l_var	:: NAME
	| l_var COMA NAME
restrict_equ_list	:: EQU equation SEMI

Table 5.8: *BNF* description of the system (2)

equation	:: binary
	| forall_list binary
binary	:: binary AND binary
	| binary OR binary
	| binary IMPLY binary
	| binary UNTIL binary
	| binary PARALLEL binary
	| binary EQUOP binary
	| binary NEQ binary
	| binary GT binary
	| binary LT binary
	| unary
unary	:: LB binary RB
	| NEXT unary
	| NOT unary
	| HENCEFORTH unary
	| EVENTUALLY unary
	| SERIAL unary
	| composed_name rest_of_name
rest_of_name	:: DOT qualification simple_or_array
qualification	:: NAME
	| TMAX
	| TMIN
composed_name	:: NAME simple_or_array
verification_decl	:: VERIFY COLUMN verify_equ_list
	| ε
verify_equ_list	:: verify_equ_list verify_equ
	| ε
verify_equ	:: EQU COLUMN equation SEMI

Table 5.9: *BNF* description of the system (3)

Operator	Associativity
HF, Next EV	Right
NOT, |. ||	Right
<, >	left
= , !=	Left
∧	Left
∨	Left
$U	Left
=>	Left

Table 5.10: Operator precendence and associativity

5.2 Semantic of TLL

In contrast with the semantic of natural languages [64], where subjective qualities are important, the semantic of programming languages is concerned with the *objective* behavior that the program has when executed by computers.

The first step that has to be taken in the specification of the semantics is to determine the *semantic domains* of the language. In the case of *TLL* the semantic domain of the language is composed of several parts:

- $\mathcal{B} = \{T, F\}$ the domain Boolean domain of *true* and *false*.
- $\mathcal{C}(\mathcal{DS})$ the class of *distributed systems models* as defined in Section 3.7.
- R the domain of *real numbers*.
- *String* the domain of strings [64].

Thus the semantic domain V of TLL is $V = \mathcal{B} \oplus \mathcal{C}(\mathcal{DS}) \oplus \mathsf{R} \oplus \mathit{String}$, where $\oplus$ denotes the sum of domains as defined in [64]. The denotation for numbers and strings is straightforward and given in Table 5.13. Although *TLL* makes a distinction between the *integers* and *real numbers*, from a mathematical point of view the set of integers is included in the set of reals and thus we will consider only the latter one. In the following discussion the emphatic brackets ($\lfloor\!\lfloor$ $\rfloor\!\rfloor$) will

Terminal Symbol	Keyword
CONSTANT	Constant, Constants
EQU	Equ
EVENT	Event, Events
EVENTUALLY	EV
GENERIC	Generic
HENCEFORTH	HF
IN	in
INIT	Init
MEMORY	Memory
MODULE	Module
NEXT	Next
OUTSPEC	Outspec, Outspecs
RESTRICT	Restrict
SYSTEM	System
STATE	State, States
TMIN	TMin
TMAX	TMax
TVARIABLE	TVariable
TTIME	TTime
VERIFY	Verify

Table 5.11: Keywords

Terminal Symbol	String Equivalence
EQUOP	=
SEMI	;
LC	{
RC	}
LB	(
RB	)
LP	[
RP	]
COMA	,
IMPLY	=>
UNTIL	$U
INITSTATE	*
DOT	.
DOTDOT	..
COLUMN	:
LT	<
GT	>
NOT	!
OUTVAL	->
PARRALEL	||
SERIAL	|
AND	∧
OR	∨

Table 5.12: Operators and other symbols

be used to separate the syntax of *TLL* from the semantics, to help avoid confusions when the language uses the same symbol for an operation as the mathematical domain [64].

$\mathcal{N}$: Numeral $\longrightarrow$ R
$\mathcal{S}$: Character-String $\longrightarrow$ *String*

Table 5.13: Denotation for numbers and strings

The semantics of modules is given over the notion of *sequential process* described in Section 3.6. Each module in *TLL* represents the description of a *P-automata*

$$\mathcal{P} = (T, E_p, Q_p, Y_p, \delta_p, \beta_p, q_{0p}, F_p, \{LOC, OP, V_L, V_R\})$$

Since our point of interest is related to reactive systems the set of final states is considered empty and does not appear in *TLL* . Hence, let $\mathcal{M}$ be the semantic function that relates a module description with its equivalent in $\mathcal{C}(\mathcal{P})$, the class of all sequential processes (Table 5.14).

$\mathcal{M}$: *Modules* $\longrightarrow$ $\mathcal{C}(\mathcal{P})$

Table 5.14: Denotation for modules

Within a module, there are several types of constructions, for which we have to provide a semantics: the module declarations that consist of state, event, memory and output declarations, the output specifications and the equations that present the dynamics of the module. For the state declaration the semantics is straightforward. Let *Mod* be a module description in *TLL* and $\mathcal{P} = \mathcal{M}[Mod]$ its corresponding denotation in the domain of $\mathcal{C}(\mathcal{P})$. Also, let $\mathcal{P} = (T, E_p, Q_p, Y_p, \delta_p, \beta_p, q_{0p}, F_p, \{LOC, OP, V_L, V_R\})$ be the *sequential process* corresponding to *Mod*. Then the denotations for the module declarations are given in Table 5.15. Each state, event or output that is part of an array declaration is considered different. The index of the array is the index of the state, event or output in their respective sets.

The denotation for the **Memory** declaration is given by the memory part of the sequential process. If A_M is the DEA (or DEAs) that make the memory structure of $\mathcal{P}$, then the value of $\mathcal{M}$ for the **Memory** declaration is given by Table 5.15. Each timed event has implicitly attached to it a time variable that keeps track of the moment when the event appeared.

$$\mathcal{M}\lfloor State\rfloor \triangleq Q_c$$
$$\mathcal{M}\lfloor s[i]\rfloor = q_{\mathcal{M}\lfloor i\rfloor} \in Q_c$$
$$\mathcal{M}\lfloor *state\rfloor \triangleq q_0 \in Q_c$$
$$\mathcal{M}\lfloor Event\rfloor \triangleq E_p$$
$$\mathcal{M}\lfloor e[i]\rfloor = e_{\mathcal{M}\lfloor i\rfloor} \in E_P$$
$$\mathcal{M}\lfloor tvar\rfloor \triangleq t \in T$$
$$\mathcal{M}\lfloor Output\rfloor \triangleq Y_P\ \mathcal{M}\lfloor o[i]\rfloor = y_{\mathcal{M}\lfloor i\rfloor} \in Y_P$$
$$\mathcal{M}\lfloor Memory\rfloor \triangleq A_M$$

Table 5.15: Denotations for module declarations

The function δ is given by the temporal logic equations that describe the dynamics of the module. The equations can be written using the declared constants, states, events and memory locations. If the module is a DEA as is the case for the communication channel an expression as that of Table 5.16 is sufficient for the description of δ. For sequential processes the equations can also contain expressions with memory locations.

$$\mathcal{M}\lfloor st \wedge ev \longrightarrow newst\rfloor \triangleq \textbf{if } (\delta(\mathcal{M}(st), \mathcal{M}(ev), t) = \mathcal{M}(newst)) \textbf{ then True else False}$$

Table 5.16: Denotation for module dynamics

The *output specifications* are the *TLL* correspondent of the set of output functions. An output function $\beta : Q \longrightarrow Y$ is defined on Q the set of states and takes values in Y. Such a function can be described by considering all the pairs (q, y) where q is a state and y an output value. An **Outspec** of the form *state* $- >$ *outval* has therefore the following denotation: $\mathcal{M}\lfloor state- > outval\rfloor \triangleq (\mathcal{M}\lfloor state\rfloor, \mathcal{M}\lfloor outval\rfloor)$.

The second part of a *TLL* specification is the *system description*. The semantics of the *system description* is given over the domain $C(\mathcal{DS})$ as presented in Table 5.17.

System description has three main parts:

- Declarations of the *objects* that compose the system. The declarations are divided into three parts. The set of processes that make the distributed system is declared using **Plant** and

$$\boxed{\mathcal{SYS} : System \longrightarrow \mathcal{C}(\mathcal{DS})}$$

Table 5.17: Semantic function for system description

Controller declarations. The set of communication channels is specified using a **Channel** declaration. All the objects that appear in a **Channel** specification have to be DEAs. Therefore their module declarations *cannot* contain **Memory** declarations.All the objects that are defined are considered to run in parallel unless, the *Connection* declaration describes them otherwise. The connections describe the input output relations between the objects that are part of the system.

- Specification of restrictions. Not all of the states that are in the Cartesian product of the state spaces are desirable. Those that are undesirable are described in the restrictions. The restrictions are specified by temporal logic equations between the states of components of the plant.
- Verification part. This part is used to verify properties that are expressible in temporal logic against the system specification.

The denotation of *objects* is given over the DEAs that compose a $\mathcal{C}(\mathcal{DS})$. If *System* is the description and $\mathcal{DS}_1$ it's semantic equivalent, let *Obj* be an object in *System* and A its corresponding DEA in $\mathcal{DS}_1$, then: $\mathcal{SYS}[Obj] \triangleq A$.

The *Generic* declaration implements the logical connective "for all" ($\forall$). There can be several instances of the same module in a system description. Let S be the set of all objects that are instances of a module description M and $\mathcal{S}$ the set of the corresponding DEAs. Then the declaration: *Generic M(x)* has the denotation presented in Table 5.18.

$$\boxed{\mathcal{SYS}[GenericM(x)] \triangleq \forall x \in \mathcal{S}}$$

Table 5.18: Semantics of Generic declaration

The *forall* construction is a restrictive implementation of the logical qualifier $\forall$. The domain over which it can be defined is restricted only to finite subsets of integers. For the unary operators

the semantics is given by the function $\mathcal{MO}$ defined in Table 5.19, while the semantics of the binary ones is given by the function $\mathcal{DO}$ defined in Table 5.20.

$\mathcal{MO} : UNARY - OPERATOR \longrightarrow (V \longrightarrow V)$
$\mathcal{MO}\lfloor Next \rfloor \triangleq \bigcirc$
$\mathcal{MO}\lfloor ! \rfloor \triangleq \neg$
$\mathcal{MO}\lfloor HF \rfloor \triangleq \Box$
$\mathcal{MO}\lfloor EV \rfloor \triangleq \Diamond$
$\mathcal{MO}\lfloor \| \rfloor \triangleq \|$

Table 5.19: Semantics of unary operators

$\mathcal{DO} : BYNARY - OPERATOR \longrightarrow (V \otimes V \longrightarrow V)$
$\mathcal{MO}\lfloor \wedge \rfloor \triangleq \wedge$
$\mathcal{MO}\lfloor \vee \rfloor \triangleq \vee$
$\mathcal{MO}\lfloor => \rfloor \triangleq \rightarrow$
$\mathcal{MO}\lfloor \$U \rfloor \triangleq U$
$\mathcal{MO}\lfloor \| \rfloor \triangleq \|$
$\mathcal{MO}\lfloor > \rfloor df >$
$\mathcal{MO}\lfloor < \rfloor df <$

Table 5.20: Semantics of binary operators

As mentioned earlier the module is the fundamental data type in *TLL* . After the parsing of a module, a state-event graph is constructed from the module description. This graph is verified for reachability in order to ensure that the description is valid from this point of view. If any problems are detected the errors are reported and the program stops. This is relatively simple for DEAs but there is the obvious problem of the state explosion for the sequential processes were the state of the process is the state of A_c combined with the state of the memory. A way to alleviate this problem would be to integrate a rewriting system and to axiomatically verify the modules instead of relying on graph verification.

Chapter 6

Using TLL for Specifying Distributed Systems

The thesis would not have been complete without an examples that presents the modeling process and the specification using *TLL* . The chosen example can be found in other forms in [78, 26] and consists of modeling a telephone system. Section 6.1 presents the telephone system model that is subsequently used in Section 6.2. Telephone systems have been extensively studied at University of Ottawa [78, 26]. Therefore, the example has the advantage that the results can be compared with those existing allready.

6.1 Telephone system description

Let us first make an informal description of the studied system. For this example, we will consider a telephone system (TS) that has only basic features. This type of system, called *Plain Old Telephone System (POTS)*, has been studied before in [78, 26]. It has to be stressed out that the name refers only to the *features* that the system offers and not to the *implementation* of the system.

The model of the system is well known. A telephone system consists of two fundamental entities (Figure 6.1):

- A set of *phone sets*, which can be used in order to make calls.
- A set of *switches*, which are interconnected together and to the phones and ensure the

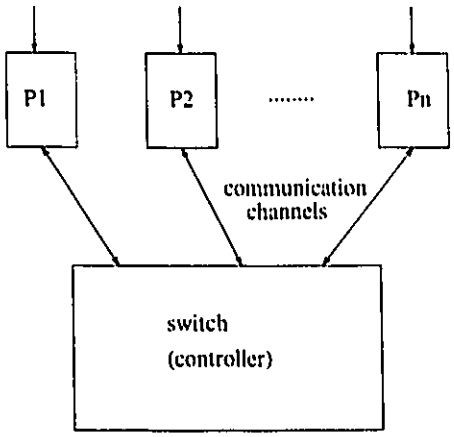


Figure 6.1: Telephone System

connections.

Since the model considers only the phones and switches, the number of users that can access a particular handset is irrelevant because we are interested only in the *events* that the phone receives.

Each phone is connected to the switch through a communication channel. There are two types of communications that take place on the channel: the actual conversation between end users and the signaling that takes place between the phone and the switch. Furthermore, each phone has two basic capabilities [26]:

- To place calls and thus, to act as a *caller*.
- It can also receive calls from another phone.

Hence, the switch acts as a *controller* between a *set* of clients (the *caller* part of the phones) and a set of resources (the *receiver* part of each phone). The set of phone sets together with the switch will be considered the the set of processes $\mathcal{P}$. The communication channels between the phone sets and the switch represent the set C from the model.

A *TS* is a distributed system because of the following reasons:

- it is spatially distributed;
- each phone and switch are independent entities that function in parallel;

Figure 6.2: Model of the phone

- the phones are connected to the switch by communication channels;
- there is cooperation towards a common goal (to make calls possible).

The concrete model of the TS, as is was seen in this section, satisfies the requirements needed in order to be classified as a distributed system. In the following section, we will try to specify the system by the means developed in this thesis.

6.2 Abstract model and specification

The next step in the modeling process is to cast the *TS* described in Section 6.1 into the abstract model of Chapter 3. Let us start by analyzing the set $\mathcal{P}$. This is composed of the set of phones and the switch. Let us model the phone set, first. At a glance it is obvious that it can be modeled as a process that accepts events and generates some outputs toward the switch. The events are received from two distinct sources:

- from the user, that can make several operations with the phone,
- from the switch itself, that sends several signals to the phone (such us dial tone, rings, busy signal etc.).

The above description can be easily cast in a DEA, whose structure we still have to determine.

For our modeling purposes, we will consider that there is only one switch in the system. This is, usually, not true in practice where a set of switches are connected together to make TS. The switching system is itself a distributed system, composed this time from the set of switches connected by communication channels.

Each phone set communicates with the switching system through a communication channel. Let us denote the set of all phones that are part of a certain system $\mathcal{S}$ by P. The actual communication protocol differs between different phone systems and is in general implementation

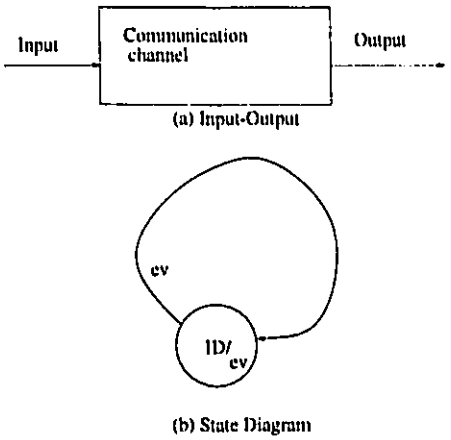


Figure 6.3: Model for the communication channel

dependent. For our purposes we will consider the simplest possible communication scheme (Figure 6.3). In this case, the bidirectional communication channel between the phone set and switch is modeled by two unidirectional channels. The communication protocol is modeled as a DEA $C = (T, Q, E, Y, \delta, \Gamma, q_0)$. The set of states Q is made up of only one state. The set of outputs Y is the same as the set of events E and the transition time is considered to be zero. Thus, this is the same as connecting the outputs of one process directly to the inputs of the other process.

The switching system is also modeled as a *sequential process*. Let us denote the controller of the system, which in this case is the switching system, by C. Thus, by Definition 3.13, the set $\mathcal{P}$ formed by $P \cup \{C\}$ together with $\mathcal{C}$ (the set of communication channels) forms a distributed system $\mathcal{DS}$ by Definition 3.13.

Let us now investigate the structure of the DEA that models the phone set. The input-output structure of the phone set model is visible in Figure 6.2. The set of *user events* is made out of the following events: *pu* (pick up the phone), *hu* (hang up the phone), *dial[number]* (dial a number). The set of events that the phone receives from the switch is composed of the followings: *dt* (dial tone), *rb* (ring back to signal that the dialing was ok), *rp* (ring a phone to signal that someone wants to talk), *bs* (busy signal, the other phone is occupied) *rs* (the ringing stops, signaling that the caller had gave up) and *t* (the actual talk). The phone set is viewed as an entity, as opposed to the view expressed in [26] were the model was divided into two parts, one for the sender and one for the receiver. Our view looks more natural and does not seem to complicate the model

since both the sending part and the receiving part are fairly simple and they are disjunct: if someone uses the phone to place a call it cannot also receive a call at the same time.

Due to the simplicity of the modeled object, the notion of DEA is powerful enough to model the phone part of the system. Let us denote the resulting DEA by $Ph = (T, Q, E, Y, \delta, \Gamma, q_0)$. The set of events $E = E_u \times E_c$ where $E_u = \{pu, hu\} \cup \{dial_i \mid \forall i \in Num\}$ and $E_c = \{dt, rp, rs, bs\} \cup \{rb_i, t_i \mid \forall i \in Num\}$. Each number that can be dialed has been modeled as a separate event, since, in our model, more than two phones can be connected to the same switch. The set of outputs represents the collection of signals that the phone sends to the switch. This set has the following structure: $Y = \{pu, hu\} \cup \{dial_i \mid \forall i \in Num\}$. Thus, in this case $Y = E_u$ since all the events that the user generates have to be send to the switch. The set of states of our system has several elements, $(Q = \{ID, OH, WD, IU, PA, WA\} \cup \{DC_i, WT_i, T_i \mid \forall in \in Num\})$, whose mnemonics are explained below.

- ID - idle. The phone is on hook and it does not ring.
- OH - off hook. The user picked up the phone and waits for the dial tone.
- WD - wait to dial. The tone was received by the phone.
- DC_i - dial complete for number i. The number i was dialed.
- IU - in use. The number that was dialed is in use.
- WT_i - wait to talk with number i. The dialed number is not in use.
- T_i - talk to number i. The call get through.
- PA - pick the receiver and answer the call.
- WA - wait to answer. The phone is on hook and rings.

Sisiruca [78] has only 5 states in his model. From his DC state [PHONE(x)W] in case that the call does not get through, his model makes a transition back to the idle state. This is incorrect, since, even if the called party does not answer no one can call the caller until it places the handset on hook. The graph of our model is represented in Figure 6.4.

The temporal logic description of the dynamics of the phone is given in Table 6.1. Subsequently, the logical equations are translated into a TLL specification. As it can be seen in Table 6.2 the specification set is quite small.

$$\Box[ID \wedge pu \rightarrow \bigcirc OH]$$
$$\Box[ID \wedge rs \rightarrow \bigcirc WA]$$
$$\Box[OH \wedge hu \rightarrow \bigcirc ID]$$
$$\Box[OH \wedge dt \rightarrow \bigcirc WD]$$
$$\forall i, \Box[WD \wedge dial_i \rightarrow \bigcirc DC_i]$$
$$\Box[WD \wedge hu \rightarrow \bigcirc ID]$$
$$\forall i, \Box[DC_i \wedge bs \rightarrow \bigcirc IU]$$
$$\forall i, \Box[DC_i \wedge hu \rightarrow \bigcirc ID]$$
$$\forall i, \Box[DC_i \wedge rb_i \rightarrow \bigcirc WT_i]$$
$$\forall i, \Box[WT_i \wedge t_i \rightarrow \bigcirc T_i]$$
$$\forall i, \Box[WT_i \wedge hu \rightarrow \bigcirc ID]$$
$$\forall i, \Box[T_i \wedge hu \rightarrow \bigcirc ID]$$
$$\Box[WA \wedge rs \rightarrow \bigcirc ID]$$
$$\Box[WA \wedge pu \rightarrow \bigcirc PA]$$
$$\Box[PA \wedge hu \rightarrow \bigcirc ID]$$

Table 6.1: Temporal logic description of phone operations

```
@ Specification   of the phone
Constant          NUMBER = 100;
Module            Phone [LIn(ce,ue);LOut(co)]
Event             ce(dt,rb[NUMBER],rs,bs,t[NUMBER]),ue(dial[NUMBER],pu,hu);
Output            co(number[NUMBER],hur,pur);
{
                  State *ID, OH, WD, DC[NUMBER], IU;
                  State WT[NUMBER], T[NUMBER], PA, WA;
                  Outspec ID -> hur,OH -> pur, PA -> pur;
                  Outspec forall i in [1..NUMBER] DC[i] -> number[i];
                  TTime = 1;

                  Equ ID ∧ pu => OH;
                  Equ ID ∧ rs => WA;
                  Equ OH ∧ hu => ID;
                  Equ OH ∧ dt => WD;
                  Equ forall i in [1..NUMBER] WD ∧ dial[i] => DC[i];
                  Equ WD ∧ hu => ID;
                  Equ forall i in [1..NUMBER] DC[i] ∧ bs => IU;
                  Equ forall i in [1..NUMBER] DC[i] ∧ hu => ID;
                  Equ forall i in [1..NUMBER] DC[i] ∧ rb[i] => WT[i];
                  Equ forall i in [1..NUMBER] WT[i] ∧ t[i] => T[i];
                  Equ WT ∧ hu => ID;
                  Equ T ∧ hu => ID;
                  Equ WA ∧ rs => ID;
                  Equ WA ∧ pu => PA;
                  Equ PA ∧ hu => ID;
}
```

Table 6.2: TLL specification of the phone

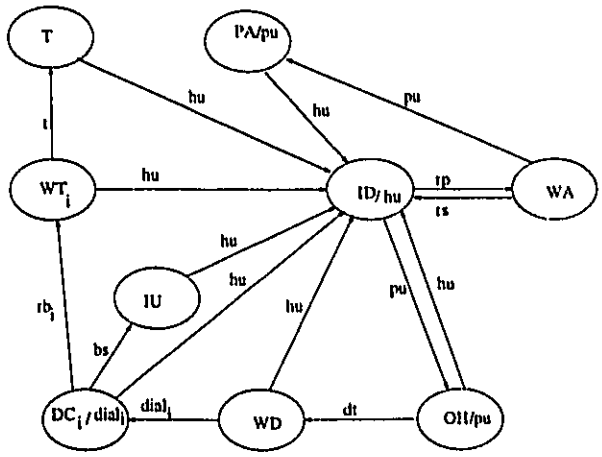


Figure 6.4: State transition diagram for the phone

The set of restrictions specifies the state space of the system that is not wanted. For example if a user picks up the handset of the phone number i and dials the number of phone j, if phone j is in use then phone i cannot be in the *wait to talk* state. Furthermore, a phone cannot be in *talk* mode with another phone unless its peer is in the PA state.

Let P_S be the set of phones that are part of our TS and $num = CARD(P_S)$. Let us also consider the interval $Int = [1..num]$ and the variables $i, j \in Int; P_i, P_j \in P_S$. We can write the restrictions described above with the temporal logic formulas presented in Table 6.3.

$\forall i,j,k \in Int, WT[i](P_j) \wedge (i \neq j) \rightarrow$	$\neg(OH(P_i) \vee DC[k](P_i) \vee T[k](P_i)$
	$\vee IU(P_i) \vee WD(P_i) \vee TP(P_i))$
$\forall i,j,k \in Int, T[i](P_j) \wedge (i \neq j) \rightarrow$	$\neg(WT[k](P_i) \vee DC[k](P_i) \vee WD(P_i)$
	$\vee OH(P_i) \vee IU(P_i) \vee WA(P_i))$

Table 6.3: Temporal logic descriptions of the restrictions

The specification for the switch has to take into account the fact that there are several module instances that will be interfaced with it. To ensure a proper functioning of the TS the switch will maintain some information about the *state* of the phones. To do this the switch needs a memory zone and thus, the concept of DEA is not sufficient to model it. The set of events for the controller is $E_C^n = E_C \times ... \times E_C$, where $E_C = \{pur, hur\} \cup \{number_i \mid i \in Int\}$. The set of output values is

also a Cartesian product, $Y_C^n = Y_C \times ... \times Y_C$, where $Y_C = \{dt, rp, rs, bs\} \cup \{rb_i, t_i \mid i \in Int\}$. Let us denote by $PH = \{ph_i \mid i \in Int\}$ the set of data stored by the controller. In this case Table 6.4 presents the temporal logic equations of the dynamics of the controller.

$\forall i \in Int, hur_i \rightarrow (ph_i = ID)$
$\forall i,j \in Int, number_{i,j} \wedge (ph_j \neq ID) \rightarrow (ph[i] = IU)$
$\forall i,j \in Int, number_{i,j} \wedge (ph_j = ID) \rightarrow (ph[j] = WA) \wedge (ph[i] = WT_j)$
$\forall i \in Int, pur_i \wedge (ph_i = ID) \rightarrow (ph_i = WD)$
$\forall i \in Int, pur_i \wedge (ph_i = WA) \rightarrow (ph_i = PA)$
$\forall i,j \in Int, pur_i \wedge (ph_j = WT_i) \rightarrow (ph_j = T_i)$
Start: $\forall i \in Int, (ph_i = ID)$

Table 6.4: Controller Specification

The temporal logic formulas are easily translatable into a *TLL* specification.The specification includes an array of memory cells (ph), that maintain information about the state of the phones. Since the individuals members of the *plant* are DEAs that do not have timed events, a single variable per phone is enough to maintain its state. As it can be observed in Table 6.5 the module has multiple inputs and outputs. Since each input set is a Cartesian product of the same set, the inputs are declared to be identical. Evidently, the output set is also a Cartesian product of identical sets and hence, the output lines are declared to be identical.

The last part in the *TLL* program is represented by the *system specification.* As presented in the previous chapter, the specification of the system starts (Table 6.6) with a series of declarations that describe the structure of the system. Here, we have the set of phones that represents the *plant* of the system, and the switch which in this case is the controller.

The declarations are followed by a description of the connections that take place between the *objects* that compose the system. Only two such equations are needed in order to describe the way in which a particular phone is connected to the switch. Finally the set of restrictions that we impose on the uncontrolled behavior of the plant is described.

The pieces of *TLL* specifications that describe the system can be put together in a program that can be run through a compiler that implements the language. Depending on the properties that the compiler can check the system can be verified against those properties.

```
@ Switch    specification
Constants   NUMBER = 100, ID = 1, OH, WA , WT[NUMBER];
Constants   IU, T[NUMBER],PA;
Module      Switch [LIn(ci[NUMBER]);LOut(co[NUMBER])]
Events      ci(pur,hur,number[NUMBER]);
Outputs     co(dt,rp,rb[NUMBER],bs,t[NUMBER]);
{
            Memory ph[NUMBER];
            Init forall i in [1..NUMBER] (ph[i] = ID);
            Outspec forall i in [1..NUMBER] ((ph[i] = OH)- > dt[i]);
            Outspec forall i in [1..NUMBER] (ph[i] = IU)- > bs[i]) ;
            Outspec forall i,j in [1..NUMBER] (ph[i] = WT[j])- > rb[i][j]);
            Outspec forall i in [1..NUMBER] (ph[i] = WA)- > rp[i]);
            Outspec forall i in [1..NUMBER] (ph[i] = T[j])- > t[i][j]);
            TTime = 1;

            Equ forall i in [1..NUMBER] hur[i] => (ph[i] = ID);
            Equ forall i,j in [1..NUMBER]
            number[i][j] ∧ (ph[j]! = ID) => (ph[i] = IU);
            Equ forall i,j in [1..NUMBER]
            number[i][j] ∧ (ph[j] = ID) => (ph[i] = WA) ∧ (ph[j] = WT) ;
            Equ forall i,j in [1..NUMBER] (pur[i] = ID) => (ph[i] = WD);
            Equ forall i,j in [1..NUMBER] pur[i] ∧ (ph[i] = WA) => (ph[i] = PA);
}
```

Table 6.5: TLL specification of the switch

@ Switch *specification*

System {

Phone p[NUMBER];

Switch sw;

Controller sw;

Connections:

Equ forall i **in** [1..NUMBER] **LIn**(sw(ci[i])) = **LOut**(p[i](co));

Equ forall i **in** [1..NUMBER] **LIn**(p[i](ce)) = **LOut**(sw(co[i]));

Restrictions:

Equ forall i,j,k **in** [1..NUMBER]

$p[j].WT[i] \wedge (i! = j) =>!(p[i].oh) \wedge !(p[i].DC[k])$

$\wedge !(p[i].T[k]) \wedge !(p[i].IU) \wedge !(p[i].WD) \wedge !(p[i].TP)$;

Equ forall i,j,k **in** [1..NUMBER]

$p[j].T[i] \wedge (i! = j) =>!(p[i].WT[k]) \wedge !(p[i].DC[k])$

$\wedge !(p[i].WD) \wedge !(p[i].OH) \wedge !(p[i].IU) \wedge !(p[i].WA)$;

}

Table 6.6: TLL specification of the system

Chapter 7

Conclusions and future directions

The present thesis was concerned with the abstract modeling of distributed systems. This approach models DS as DES and uses the resulting abstraction to build several models and a logical language. The logical language was cloned on Mana and Pnueli's temporal logic. Temporal logic, as it can be seen in the thesis, has a natural ability to describe dynamical systems. The use of temporal logic for the description of DES and distributed systems is well known, but in this thesis we made a step forward and modeled the distributed systems as DES. This has the definite advantage of making the link between fields by applying to DS well known results from DES.

The debut of the work is marked by a presentation of the motifs. Chapter 2 presents the current state of the research in the field as it can be seen in the published materials related with the subject. A quick overview of universal algebra notions, which are used in Chapter 3, is also given at the end of the second chapter.

Chapter 3 represents the foundation of the thesis and is dedicated to the study of a suitable systemic model for distributed systems. This approach is unique, a similar concept is not yet available. The starting point in our study is the quest for a concrete model that suits our need. Once the model is adopted, the next step is to construct an abstract model for distributed systems. This is done in several steps. First a base model, derived from Kalman's [38] notion of system, is described. This is a timed model with a dense set of time values. The base model is subsequently studied from an algebraic point of view. The universal algebra is a handy instrument for studying the properties of the DDES and DEA. It also enables us to make the connection with a logical framework. Three fundamental types of compositions of the base model are also defined. From

this base model a model for the notion of *sequential process* is investigated. By combining the *sequential process* with the concrete model the result is an abstract model for distributed systems.

In Chapter 4 a logical framework, derived from temporal logic, is presented. While the syntax of our logic does not present significant differences with that of Mana and Pnueli, the semantics is different since it is based on the abstract models developed in Chapter 3. Two new operators are introduced in order to better present the parallel nature of the distributed systems. The end of the chapter is marked by the presentation of an axiomatic system, which can be used with the logic.

Chapter 5 makes the transition from theory to practice. It seems almost natural to try and automate the specification process by using computers. The syntax and semantics of a language, that can be implemented in a compiler, is described in the sections of this chapter. The choice for a text based programming language was made because we consider that it is easier to represent big specification in such a language than in a graphical environment. In my opinion, graphical programming environments have only limited applicability, due to the relatively small amount of information that can be displayed at once. For example, the environment presented in [78] filled the screen after only 30 states, thus making the results unreadable. It also seems more easily to type a program than to use the mouse in order to construct each rule separately.

The last chapter of the thesis is concerned with an example of a specification, using the language described in Chapter 5. The example chosen was treated in detail by Faci [26], but the approach to modeling is different in this work: while he tries to model only a pair of phones, we try to globally model a set of phones together with their respective switch. The *TS* is proved to be a distributed system in the way defined in Chapter 3. The specification of the *TS* is presented in both, the temporal logic of Chapter 4 and *TLL* the programming language described in Chapter 5.

The work started in this thesis has many possible continuations. The most important, and the most difficult I might say, line of research that can followed is related to the study of the base abstract model (DDES/DEA) presented in Chapter 3. Several systemic properties *reachability, controllability, observability and stability* are yet to be studied from an algebraic point of view. A Lin type of description for the *plant - controller* combination in the case of an hierarchy of controllers can also be sought. An abstract model of distributed systems that takes into consideration the variability in the number of running processes that run at a certain moment

can be investigated.

The logical framework of Chapter 4 can be refined by improving the axiomatic. This can be done by considering a set of axioms that take into account systemic properties, such as those mentioned above. We can than prove the correctness, at least in from a systemic point of view, in the axiomatic system. Thus, the verification step will not have to resort to other methods (such as state graph verification) for that.

The language presented in Chapter 5 is probably prone to more controversy, since, as in any design activity, some decisions that affect the final form, without modifying the content, can be made. To be more specific, several major examples should be described in the language in order to verify the viability of the syntax. And since the semantics is related to the theoretical framework that was previously presented, any change in the theory will have a direct impact on the language. In order to automate the verification process, once the axiomatic system has been perfected, a theorem prover should be integrated into the system. From an early look into the area, since our abstract model is algebraic, an equational rewriting system such as the one used by *OBJ* [25] seems promising. Finally, since operational semantics of temporal logic programs are known, the integration of a code generator with the software framework also looks like a promising perspective. To achieve this, an operational semantics isomorphic with the semantics that already exists should be investigated.

This thesis accomplished several things. In the end, let us enumerate them:

- A new theoretical framework for modeling distributed systems, that applies a systemic point of view, has been presented in Chapter 3. Three types of compositions over the base abstract model are presented and using them a model for distributed systems was generated.

- Classical temporal logic was modified, in chapter 4, by adding two new operators that seem better suited to represent the true parallelism of distributed systems. The semantic of the logic is given over the abstract model of distributed systems developed in Chapter 3.

- A new specification language is presented in Chapter 5 of the thesis. Using this language a simple telephone system is modeled and specified.

Bibliography

[1] Martin Abadi and Leslie Lamport. An old-fashioned recipe for real-time. In *Real-Time: Theory in Practice*, volume 600, Lecture Notes in Computer Science, pages 1–27. Springer Verlag, 1992.

[2] Jiri Adamek and Vera Trnkova. *Automata and Algebras in Categories.* Kluwer Academic Publishers, 1989.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, techniques and tools.* Addison Wesley, 1988.

[4] J. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26:832–843, 1983.

[5] R. Alur and D. L. Dill. The theory of timed automata. In *Real-Time: Theory in Practice*, volume 600, Lecture Notes in Computer Science, pages 45–73. Springer Verlag, 1990.

[6] M. A. Arbib. *Theories of Abstract Automata.* Series in Automatic Computation. Prentice Hall, Englewood Cliffs, N.J., 1969.

[7] M. A. Arbib and E. G. Manes. *Arrows, structures and functors: The categorical imperative.* Academic Press, 1975.

[8] Howard Barringer, Ruurd Kuiper, and Amir Pnueli. Now you may compose temporal logic specifications. In *Proc. 16th Ann. ACM Symmp. on Theory of Computing*, pages 51–63, 1984.

[9] B.A. Brandin and Murray W. Wonham. Supervisory control of timed discrete-event systems. *IEEE Transactions on Automatic Control*, AC-39(2):329–343, February 1994.

[10] R. J. Buchi. *Finite Automata, Their algebras and grammars.* Springer-Verlag, 1989.

[11] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *VLSI 91*, Edinburgh, Scotland, 1990.

[12] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference*, 1990.

[13] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS*, 1990.

[14] C. C. Chang and H. J. Keisler. *Model theory.* North-Holland, 1980.

[15] E. M. Clarke and O. Grumberg. The model checking problem for concurrent systems with many similar processes. In *Temporal logic in specification*, volume 398, Lecture Notes in Computer Science, pages 188–202. Springer Verlag, 1989.

[16] E.M. Clarke and E. Allen Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981.* Springer-Verlag, 1981. volume 131 of *Lecture Notes in Computer Science.*

[17] E.M. Clarke, E. Allen Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[18] P. Cousot. *Handbook of Theoretical Computer Science*, chapter Methods and logics for proving programs, pages 843–994. MIT Press, 1990.

[19] H.D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic.* Springer-Verlag, 1984.

[20] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer-Verlag, 1985.

[21] E. Allen Emerson. *Handbook of Theoretical Computer Science*, chapter Temporal and Modal Logic, pages 995–1073. MIT Press, 1990.

[22] E. Allen Emerson and J.Y. Halpern. "sometimes" and "not never" revisited: On branching versus liniar time. In Texas Austin, editor, *Proc. 10th Annual ACM Symp. on Principles of Programming Languages*, pages 127–140, 1983.

[23] E. Allen Emerson and J. Srinivasan. Branching time temporal logic. In *Linear time, branching time and partial order time in logics and Models of concurrency*, volume 354, Lecture Notes in Computer Science, pages 123–173. Springer–Verlag, 1989.

[24] H.A. Enderton. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.

[25] J.A. Goguen et al. Introducing obj. Technical Report SRI-CSL-92-03, Computer Science Laboratory, March 1992.

[26] M. Faci. *Detecting feature interactions in telecommunications systems design.* PhD thesis, University of Ottawa, 1995.

[27] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. Technical report, Department of Computer Science, University of Edinburgh, 1990.

[28] M. Harrison. *Introduction to formal language theory.* Addison-Wesley, 1978.

[29] Matthew Hennessy. *Algebraic Theory of Processes.* The MIT Press, Massachusetts, 1988.

[30] Matthew Hennessy. Observing processes. In *Linear time, branching time and partial order time in logics and Models of concurrency*, volume 354, Lecture Notes in Computer Science, pages 173–201. Springer–Verlag, 1989.

[31] T.A. Henzinger, Zohar Manna, and Amir Pnueli. Timed transition systems. In *Real-Time: Theory in Practice*, volume 600, Lecture Notes in Computer Science, pages 226–252. Springer Verlag, 1990.

[32] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 10(12):576–580, 1969.

[33] C. A. R. Hoare. *Communicating sequential processes.* Prentice-Hall, 1985.

[34] G. Hommel. Language constructs for distributed systems. In *Distributed systems*, volume 190, Lecture Notes in Computer Science, pages 287–337. Springer Verlag, 1985.

[35] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages and computation.* Addison-Wesley, 1979.

[36] Dan Ionescu. Designing supervisor for real-time systems. In Teodor Rus and Charles Rattray, editors, *Theories and experiences for real-time system development*, pages 103–128, 1994.

[37] Lin J.Y. *A Temporal Logic Approach to the Analysis and Synthesis of Discrete Event Systems.* PhD thesis, University of Ottawa, 1993.

[38] R. Kalman, P. Falb, and M.A. Arbib. *Mathematical Theory of Dynamic Systems.* Academic Press, New York, 1969.

[39] S. Katz and D. Peled. Interleaving set temporal logic. In *Temporal Logic in Specification*, volume 398, Lecture Notes in Computer Science, pages 21–43. Springer Verlag, 1987.

[40] H. Kopetz and P. Verissimo. *Distributed Systems*, chapter Real time and dependability concepts, pages 411–447. ACM Press, 1993.

[41] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2:255–299, 1990.

[42] Ron Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic.* LNCS 651. Springer Verlag, 1992.

[43] F. Kroger. *Temporal Logic of Programs.* Springer Verlag, New York, 1987.

[44] C. Lambiri and D. Ionescu. A temporal logic language for distributed environments. In *Proceedings of CCECE 94*, September 1994.

[45] Leslie Lamport. Temporal logic of actions. Technical report, Digital Equipment Corporation, SRC, 1991.

[46] Leslie Lamport. Verification and specification of concurrent systems. In *A Decade of concurrency : reflections and perspectives*, volume 803, Lecture Notes in Computer Science, pages 347–375. Springer Verlag, 1994.

[47] Leslie Lamport and Nancy Lynch. *Handbook of Theoretical Computer Science*, chapter Distributed Computing: Models and Methods, pages 1159–1202. MIT Press, 1990.

[48] Leslie Lamport, R.Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 3(4):382–401, 1982.

[49] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specifications. In *Proc. 12th ACM Symmp. on Principles of Programming Languages*, pages 97–107, 1985.

[50] J.Y. Lin and D. Ionescu. A generalized temporal logic approach for control problems of a class of nondeterministic discrete event systems. In *Proc. 29th IEEE COnf. Decision and Control*, pages 3340–3345, Honolulu, Hawaii, Dec. 5–7 1990.

[51] J.Y. Lin and D. Ionescu. A reachability synthesis procedure for discrete event systems in a temporal logic framework. to appear in IEEE Trans. Systems, Man and Cybernetics, September 1994.

[52] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Prog. Lang. and Systems*, 6(1):63–93, 1984.

[53] Zohar Manna and Amir Pnueli. Proving precedence properties: the temporal way. In *Automata, Languages and Programming*, volume 154, Lecture Notes in Computer Science, pages 491–512. Springer Verlag, 1983.

[54] Zohar Manna and Amir Pnueli. Verification of concurrent programs: A temporal proof system. In *Foundations of Computer Science IV*, volume Mathematical Centre Tracts 159, pages 163–255. Mathematish Centrum, Amsterdam, 1983.

[55] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems.* Springer-Verlag, 1992.

[56] Zohar Manna and A. Pnuelli. The anchored version of temporal logic. In *Linear time, branching time and partial order time in logics and Models of concurrency*, volume 354, Lecture Notes in Computer Science. Springer–Verlag, 1989.

[57] H. McGuire, Z. Manna, and R. Waldinger. Annotation-based deduction in temporal logic. In *Temporal logic*, volume 827, Lecture Notes in Computer Science, pages 430–445. Springer Verlag, 1994.

[58] K. Meinke and J. V. Tucker. *Many sorted logic and its applications.* John Wiley, 1993.

[59] E. Mendelson. *Introduction to mathematical logic.* D. van Nostrad Company, 1979.

[60] M.D. Mesarovic and Yasuhiko Takahara. *General systems theory: mathematical foundations.* Academic Press, 1975.

[61] Jose Meseguer and Joseph A. Goguen. *Algebraic methods in semantics*, chapter Initiality, induction and computability, pages 461–541. Cambridge University Press, 1985.

[62] Robin Milner. *Communication and concurrency.* Prentice-Hall, 1989.

[63] Robin Milner. *Handbook of Theoretical Computer Science*, chapter Semantics of concurrent processes, pages 1203–1242. MIT Press, 1990.

[64] P.D. Mosses. *Handbook of Theoretical Computer Science*, chapter Denotational Semantics, pages 577–629. MIT Press, 1990.

[65] B. Moszkowski. *Executing temporal logic programs.* Cambdridge University Press, 1986.

[66] Takao Murata. Petri nets: Properties, analisys and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[67] M.U.Sorensen, O.E. Hansen, and H.H. Lovengreen. Combining temporal specification techniques. In *Temporal logic*, volume 827, Lecture Notes in Computer Science, pages 1–17. Springer Verlag, 1994.

[68] Novell. *IPX Router Specification*, part number 107-000029-001 edition, October 1993.

[69] M.A. Orgun and W. Ma. An overview of temporal and modal logic programming. In *Temporal logic*, volume 827, Lecture Notes in Computer Science, pages 445–480. Springer Verlag, 1994.

[70] J.S. Ostroff. *Temporal Logic for Real-Time Systems.* John Wiley, New York, 1989.

[71] J.S. Ostroff and Murray W. Wonham. A framework for real-time discrete event control. *IEEE Transactions on Automatic Control*, 35:386–397, 1990.

[72] A. Pasztor and I. Sain. A streamlined temporal completness theorem. In *CSL'89: 3rd Workshop on Computer Science Logic*, volume 440, Lecture Notes in Computer Science, pages 322–336. Springer Verlag, 1990.

[73] S.S. Pinter and P. Wolper. A temporal logic for reasoning about partially ordered computations. In *Proc. 3rd ACM Symmp. on Principles of Distributed Computing*, pages 28–37, 1984.

[74] Amir Pnueli. The temporal logic of programs. In *Proceedings of the Eighteenth Symposium on the Foundations of Computer Science*, pages 46–57, 1977.

[75] Arthur Prior. *Past, present and future.* Clarendon Press, Oxford, 1967.

[76] F. Schneider. *Distributed Systems*, chapter What good are models and What models are good?, pages 17–27. ACM Press, 1993.

[77] M. D. Schroeder. *Distributed Systems*, chapter A state of the art distributed system: Computing with BOB, pages 1–16. ACM Press, 1993.

[78] A. Sisiruca. An object oriented environment for simulating disrete event systems in a temporal logic framework. Master's thesis, University of Ottawa, 1994.

[79] J. F. Sogaard-Anderson, Nancy Lynch, and Butler Lampson. Correctness of communication protocols. Technical report, MIT Computer Science Laboratory, 1993.

[80] Colin Stirling. Comparing linear and branching time temporal logcis. In *Temporal Logic in Specification*, volume 398, Lecture Notes in Computer Science, pages 1–21. Springer Verlag, 1989.

[81] Colin Stirling. *Handbook of Logic in Computer Science*, chapter Modal and temporal logics, pages 478–551. Oxford University Press, 1992.

[82] Andrew Tanenbaum. *Modern operating systems.* Prentice-Hall, 1992.

[83] Johan van Benthem. Time, logic and computation. In *Linear time, branching time and partial order time in logics and Models of concurrency*, volume 354, Lecture Notes in Computer Science, pages 1–47. Springer–Verlag, 1989.

[84] Wolfgang Wechler. *Universal Algebra for Computer Scientists.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1992.