NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

An Object Oriented Interactive Simulator For Discrete Event Systems In A Temporal Logic Framework

by

Alfredo Sisirucá

A THESIS

submitted to the School of Graduate Studies and Research

in partial fulfillment of the requirement

for the degree of

Master of Applied Sciences

in

Electrical Engineering

Ottawa-Carleton Institute for Electrical Engineering

Department of Electrical Engineering

Faculty of Engineering

University of Ottawa

OTTAWA, ONTARIO K1N 6N5

ISBN 0-612-00498-8

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.

Alfredo Sisirucá

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.

Alfredo Sisirucá

ABSTRACT

As more sophisticated systems are being developed, powerful approaches for modeling their behavior and test their reliability are necessary. The research work in this thesis takes on the problem of building a Graphical Programming Environment that permits to create models of DESs in a timed teporal logic framework and simulate the DES models in real-time using an object oriented environment through the interconnection of visual symbols.

A temporal logic framework is developed to write the formal models of the temporal references of DESs. This approach is enhanced by the inclusion of a global clock variable to add real-time properties to the formal specifications of real-time DESs.

The interactive visual environment allows the programmer to activate graphical symbols by means of menu selections. The graphical symbols are grouped into classes which are eventually properly interconnected, parsed, and mapped into source code written in the timed temporal logic language.

A knowledge-based system is composed of knowledge databases (database of facts and database of rules). These databases, representing the system behavior, can be created using this tool, for which a reasoning mechanism is required. An inference engine is designed to interpret these knowledge databases.

An OO programming language is used, Objective-C. It is used throughout the design, however, when using the tool, the user does not notice the underlaynig programming language, in other words, the programming language is transparent to the user.

The Graphical Programming Environment designed in this thesis can be used to build the specifications of real-time DESs. Different knowledge databases have been created using this interactive tool for three examples to verify their behaviors, such examples are: the ABP communication protocol, the packet-switched communication protocol, and the telephone system.

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my supervisor, Dr. Dan Ionescu. He has been a constant source of support, guidance, and ideas for me. I am most appreciative of his patient assistance.

I would like to thank Dr. Teodor Rus (Iowa University). He guided me in designing the parsing mechanism and provided me with some parsers already implemeted by his research team.

I would also like to extend my gratitude to my wife, Maria Eugenia, my family and in-laws for their constant support and encouragement.

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

$\mathbf{F}$	the set of formulas.
$\mathbf{F}^*$	the set of subsets of $\mathbf{F}$.
f, p	flexible or rigid function or predicate.
D	data structure.
$\mathbf{T}$	time set.
I	set of sorts.
$\mathcal{M}$	many-sorted model.
M	the representation of a structure.
$M[t]$	represents the value assigned to the term t by M.
f^M	represents the function assigned to the function letter f by M.
P^M	represents the predicate assigned to the predicate letter P by M.
N	set of rational numbers.
Q	set of prime natural numbers.
$delta_{GlobClock}$	global clock of the system.
$delta_{clock(x)}$	other clock in the system.
T_a, T_b	time bounds.
R^+	the set of the positive real numbers.
$\mathbf{S}$	the set of states; and $s \in \mathbf{S}$.

Greek Letters

α	(also ψ, γ, φ) timing symbol.
ϵ	the null event.
ϕ	the empty sequence or the empty set.
σ	a sequence of states or events, or a trajectory.
$\sigma^{(k)}$	a sequence of states starting from s_k.

θ the cost function.

Γ a set of formulas.

κ is a constant that belongs to Q.

Special Symbols

$\neg$ the logic negation symbol.

$\vee$ the disjunction connective.

$\wedge$ the conjunction connective.

$\Rightarrow$ the conditional implication.

$\Leftrightarrow$ the biconditional implication.

$\bigcirc$ the next operator.

$\Box$ the henceforth operator.

$\Diamond$ the eventually operator.

$\leq$ the less than or equal to operator.

$<$ the less than operator.

$\forall$ quantifier symbol.

$\mathcal{U}$ the until operator.

$\cup$ the union of sets.

$\cap$ the intersection of sets.

Acronyms and Definitions

AI artificial intelligence.

ABP alternating bit protocol.

CWA closed world assumption.

DES discrete event system.

DESs discrete event systems.

DP dynamic programming.

FSM finite state machine.

GPE graphical programming environment.

GUI	graphical user interface.
ITL	interval temporal logic.
OOP	object-oriented programming.
POTS	plain old telephone system.
PSCP	packet-switched communication protocol.
PSN	packet-switched node.
PSNs	packet-switched nodes.
RTTL	real-time temporal logic.
TDES	timed discrete event system.
TDESs	timed discrete event systems.
TL	temporal logic.
TLM	temporal logic model.
TLMs	temporal logic models.
TTL	timed temporal logic.
TTLM	timed temporal logic model.
TTLMs	timed temporal logic models.
TS	telephone system.
UIDS	user interface development system.
VPE	visual programming environment.

Chapter 1

Introduction

The goal of this thesis is to develop a Graphical Programming Environment to facilitate the construction of Timed Temporal Logic Models (TTLMs) by means of icon interconnection for model development, specification, and verification of Timed Discrete Event Systems (TDESs) in a temporal logic framework. The framework is an extension of the temporal logic language designed by Lin in [36] to which the concept of time metrics has been added, thus the extended language used here is a timed-temporal logic language

As window systems become more popular, there is growing need for Graphical User Interfaces (GUIs). At the analysis stage, the interactive user friendly graphical interface was required to provide the programmer with a set of class objects chosen from a selection menu to build on the screen the temporal logic models (rules and facts) by means of interconnecting graphical symbols. In other words, the user would be able to create a knowledge data base from interconnected iconic symbols that would represent the description of the system to be simulated. Also, it required that a reasoning mechanism (inference engine) be implemented to interpret the knowledge data base created by means of using this visual tool as well as a mechanism to present the simulation outcome in an amiable and easy to understand manner.

This thesis develops a prototype graphical environment for temporal logic modelling of

TDESs simulation problems developed in the object-oriented environment of Objective-C. It also represents a continuation of the work done by Lin in [36].

Lin addresses the analysis and synthesis of DESs using a temporal logic framework. This approach provides a temporal logic model and a temporal logic language for the modelling and specification, an algorithm for reachability analysis, and a procedure for the controller design and synthesis of DESs. An optimization method has been also developed to design the controller able to generate a sequence of events which lead the system to go from the initial state to the goal state following the optimal path.

Temporal logic is proving to be useful in the area of specifying the behavior of real-time DESs [45]. Four temporal operators are implemented to write our temporal logic models. These temporal operators are: EVENTUALLY ($\Diamond$), NEXT ($\bigcirc$), HENCEFORTH ($\Box$), and UNTIL ($\mathcal{U}$). Time bounds are associated to the temporal operators and predicates and along with a time variable (clock), time is being treated less abstractly than using pure temporal operators.

Timing behavior of real-time DESs can be represented through temporal logic models. In the explicit clock approach [53], timing properties are described using a global time variable whose value increases from one state to the next. This has the advantage of not requiring new techniques for reasoning about the timing properties of systems [45]. On the other hand, the approach of having unrestricted use of a time variable, for example, to have a time variable at every single state, leads us to create new proof rules to reason about the timing properties of the system, allowing for a more elegant way of associating time bounds with temporal operators in the temporal logic models.

In what follows a general overview will be given on temporal logic as a framework for real-time DESs, real-time processes as DESs, DES simulation, visual programming, and the programming technique throughout this research, respectively; and finally, our contribution to the field will be given.

1.1 Temporal Logic

Temporal logic is a branch of formal logic which has been of particular interest to the computer science community since the pioneering work done by Amir Pnueli in the late 1970s [51, 52]. Since then, research concerned with temporal logic has become substantially broader, with many new ideas and results having immediate relevance to computer science.

DESs are systems that must satisfy timing properties. Temporal logic was introduced for reasoning about propositions whose truth or falsity values might change as a function of time. The temporal propositions consist of formulae in a many sorted first-order logic, with each atomic predicate taking some set of temporal arguments as well as a set of non-temporal arguments. The temporal arguments serve to specify the predicate's dependence on time.

Temporal logic is a formalism for reasoning about a changing world. Because the concept of time is built into the formalism it has been widely used for specification and verification of concurrent programs [37, 34] and for description of activities where the chronological order of events is central, such as, in planning and historical databases [3, 13]. This has led to a number of temporal logic programming languages [42, 19, 21, 1, 29, 43] or proposals about how temporal logic can be used for programming [22].

Temporal logic has been used widely to specify and reason about reactive systems [53]. A linear-time temporal logic describes and verifies the properties of state sequences. A branching-time temporal logic, on the other hand, is used to reason about state trees which describe a non-deterministic behavior of the system.

The areas in which temporal logic models can be used are: distributed systems, concurrent programs communication protocols, hardware specifications, database applications, artificial intelligence, computer operating systems, manufacturing systems, etc.

1.2 Discrete Event Systems (DESs)

A *Discrete Event System* (DES) is a dynamical system whose evolution in time develops as a result of the occurrences of physical events at possibly irregular time intervals, as in the case of real-time systems. The characteristic features of DESs include the following:

- Events occur at discrete times instantaneously;
- States have discrete values which may be non-numerical and symbolic;
- Systems are event-driven rather than clock driven;
- Operations of processes are non-deterministic in general;
- Systems may have internal dynamic behavior and also interact and react with their environment; and
- Processes may operate concurrently and communicate with each other.

DESs model a large variety of practical systems, including systems that have not been known as discrete event systems traditionally. For instance, manufacturing systems, computer operating systems, communication networks, data bases, etc.

Fundamental to all DES models is that the behavior of the systems is naturally described by a record or trace of certain qualitative changes, better known as events, in the system. The occurrence of an event is controlled by a set of conditions. A condition is a predicate or logical description of the states and/or the inputs of the system. Such a condition may either hold or not hold. For an event to be enabled to occur, it may be necessary that certain conditions hold. These are the preconditions of the event. Thus, the set of possible events that can occur at a given time moment from some initial time point and the set of preconditions that are defined by the event must hold before the event can occur. Events are generally assumed to occur asynchronously and instantaneously, and their effect is immediately registered.

DESs are divided into two parts: **plant** and **controller**. The plan and the controller work as two concurrent processes. The plant generates the events that have to be recognized by the controller to ensure that the unsatisfactory behavior of the plant is eliminated.

Lin in [36] mentioned several approaches that have been attempted in constructing general models but there still has been a lack of a universal framework for DESs. These approaches are basically different means for modelling and analysis of different aspects of the system behavior.

1.3 Real-Time Processes

Real-time processes are DESs in which time constraints are imposed [53]. They are concurrently running processes that must maintain ongoing cooperation among themselves and with the environment so that appropriate reactions to critical events are completed by suitable deadlines. Those critical events may be pressures rising above safe levels, the arrival of a message, the receipt of a signal, or human operators depressing buttons on a control panel. The timing constraints are real-time deadlines that must be met for safe operation. The system correctness depends not only on the logical result of the system behavior but also on the time at which the results are produced.

In the context of real-time processes, the term modelling refers to a precise behavioral description of the critical features of the system [45]. A real-time process can be modelled in terms of its two major components: *states and events*. As it was mentioned in the previous section, a state describes an activity that has duration in time and an event describes a qualitative change that may lead to an instantaneous change from one state to another.

It has long been conjectured that a formal mathematical approach would be useful in overcoming some of the difficulties in the specification, design, and implementation of complex real-time systems. Turning this conjecture into a sound practice has proved to be extremely difficult, however many benefits can be obtained by using a formal framework.

These benefits include:

- In the process of formalizing, informal requirements, ambiguities, omissions, and contradictions will often be discovered.
- The formal model may lead to a hierarchical semi-automated (or even automated) system development method.
- The formal model can be verified for correctness by mathematical methods rather than by intractable case by case testing.
- A formal verified subsystem can be incorporated into a larger system with greater confidence when it behaves as specified.
- Different designs can be compared.

Many different formal frameworks have been designed with some of the above goals. Ostroff in [45] presents a framework called ESM/RTTL. The ESM/RTTL framework is used for modelling, specifying, and verifying systems composed of real-time discrete event processes. *Extended state machines* (ESMs) are used to represent real-time discrete event processes, and *real-time temporal logic* (RTTL) is the assertion language for specification and verification.

The connection between ESMs and RTTL is made via an extension to Manna and Pnueli's notion of a fair transition system [45]. A fair transition system consists of a (possibly finite) state space, a set of transitions defining state transformations, a set of initial states, and a justice and fairness family of states.

Another formal approach is presented by Xudong in [62]. In this paper, a new methodology of integrating time predicate transition nets (a class of higher-level Petri nets) and real-time first order temporal logic is developed for specifying and verifying real-time systems. The integration of time predicate transition nets with real-time temporal logic is based on the extension of time features. The methodology adds a number of significant features for the analysis of DESs. These features are: a real-time system is specified by using

a time predicate transition net, a consistent real-time temporal logic system is derived from the time predicate transition net specification, system properties are expressed in real-time temporal logic, and system properties are verified by using the derived real-time temporal logic axioms and inference rules.

1.4 Simulation

Simulation techniques are increasingly being used to study large complex systems mainly helping in the development and optimization of such systems. The objective in constructing a model is to represent as accurately as possible those aspects of the real world system which are relevant to the analysis objectives to obtain information by observing the system performance. The information so obtained might be used to modify the system behavior. Thus, computer simulation is indeed a valuable contribution to the system design process.

As hardware technology advances, computer systems tend to adopt distributed architectures in order to more fully utilize system resources. Many of these systems demand superior real-time performance. The behavior of such real-time distributed systems becomes so complex that it is very difficult to verify the specification of the systems. However, it is highly desirable to determine if it is feasible to implement a system according to its specification. For such a feasibility study real-time simulation has shown to be a promising way to accomplish that goal.

Chang et al in [11] choose a class of telephone switching system, PBX (Private Branch eXchange), to verify the objective system, since it becomes more and more distributed and communications among processors are much more complex. Telephone switching systems are typically developed in a so-called cross-development environment operating in a host machine. In such an environment, simulation techniques have been often applied to verifying the software system.

Ozden in [48] presents a prototype graphical programming methodology for modelling and automatic interpretation of simulation problems. The user uses a high-level graphical representation formalism based on the activity-cycle diagram to define simulation models in an interactive mode.

In the literature we found another example of graphical simulations in [56]. In this paper a language-based environment for a specific visual language is generated in Visual Programmer's WorkBench (VPW) from a specification of the syntactic structure, the abstract structure, the static semantics and the dynamic semantics of the language. This framework is used both in defining the architecture of the environment and for the execution model of a telephone system.

1.5 Visual Programming

Recently, there has been a great deal of interest in systems that utilize graphics in human-computer communications, in programming, and visualizing programs and data. Visual programming and program visualization are very appealing concepts for a number of reasons. The human visual system and the human visual information processing are clearly optimized by multi-dimensional data representation. Graphical programming uses information in a format that is closer to the user's mental representation of problems, and allows data to be manipulated closer to the way objects are manipulated in real world.

Visual programming is a key to enhancing computer utilization by allowing those who are not familiar with programming languages to create programs. In addition, since the people creating programs are not necessarily programmers but specialists in their own professional domain, it is expected that visual programming may result in an yet-unpredictable improvement in software quality as well as productivity.

A single definition of visual programming might be programming with pictures. The idea of programming a computer by drawing pictures is intuitive to most engineers who are

familiar with flowcharts and block diagrams. The goal of visual programming languages is to allow the engineer to design a solution to the problem at a very high level and then directly execute the resulting picture.

The diagrams may consist of processing blocks or nodes to indicate operations on the data, and arcs or connections to indicate the flow or movement of data. The graph can be directly executed or used as input for a compiler to produce an executable program.

Computer engineers and scientists are attempting to make the process of programming more productive, and the resulting software of higher quality and reliability. From those attempts, HI-VISUAL [26] and Khoros visual programming environments [54] have resulted. The applications in these two programming, dataflow oriented, languages are built via icon interconnection where each icon is the representation of a function.

Simulations of a modelled system produce invaluable information that supports critical management and engineering decisions. Two key cases in which critical decisions must be supported are: (1) feasibility studies of proposed systems and (2) performance estimates of existing systems that cannot be studied under operational conditions because of economic, safety, political, or any other reasons.

Graphical languages or models provide a means for describing and representing systems in a precise, understandable, and cost-effective manner. In a simulation, a software tool directly interprets the model written in one of the graphical languages; hence, the simulation is closely related to the graphical representation of the modelled system. The closer the structural and behavioral characteristics of the model match the real world system, the more likely that the analysis objectives will be achieved making graphical representation an approachable tool for doing simulations.

1.6 Object-Oriented Programming

The term *object-oriented problem solving* implies that problems are solved through the development and use of relationships among objects. The five key components of the object-oriented paradigm are: object, message, class, instance, and method. These components provide a consistent thread that is woven through the analysis, design, and implementation of object-oriented software systems and all together perform the following major operations:

- Define and develop descriptions for classes representing objects that are part of the problem solution. The class description must include a way to create new instances of the class. It must also include all the messages and methods to which the class instances can respond.
- Create the objects and develop a sequence of expressions (messages sent to objects) that effect a solution to the problem.

Object oriented programming (OOP) is a way of structuring programs by defining a set of objects capturing information about corresponding entities that are of interest to the modeler. Information processing is performed through the communication between objects which send messages to each other. Object oriented problem solving paradigm is based on building a solution procedure by specifying a sequence of messages to be sent to a particular object. Implementing the object oriented paradigm for solving a specific problem can be done in four steps [50]: problem definition; identification of the domain objects; identification to the messages to be sent to those objects; and definition of a problem solving sequence of messages.

There has long been interest in applying OOP to the construction of simulation models. In fact, OOP was first supported in the SIMULA simulation language [9]. OOP is particularly appropriate for implementing hierarchical, modular simulation modelling and distributed simulation. Hierarchical, modular simulation modelling and distributed simulation are rarely done using anything other than OOP language or approach.

OOP is increasingly used in discrete event simulation. Zeigler in [63] described compatibility between OOP paradigm and discrete event simulation. In a recent work [64], the development of an approach to object oriented discrete event simulation based on the concept of hierarchical, modular model construction is described.

1.7 Research Contributions

The research work presented in this thesis has the contributions to the field summarized as follows:

- Extended the temporal logic framework and temporal logic language into a timed-temporal logic framework and timed-temporal logic language, respectively, for the analysis, synthesis, simulation, and optimization of DESs and real-time systems.
- An important effort to integrate the object-oriented paradigm with the analysis of discrete event systems in a timed-temporal logic framework. Crucial and complex information about system behavior can be encapsulated within the class object entities.
- An effort to provide an object-oriented user-friendly environment for visual programming using temporal logic reasoning in simulation tasks.
- Designed and implemented a GUI that:
  - allows for accomplishing visual programming in the above timed-temporal logic language.
  - combines the concepts of visual programming and user-friendliness to provide an interactive programming environment for simulations of DESs in a timed-temporal logic framework whose knowledge (TTLMs) is built by means of graphical icon interconnections.
  - shows the dynamic behavior of the system components on the screen as animated graphs.

- The term reusability has been widely used and implemented throughout this thesis. This frees up the user from repeating things when building system simulations.

Chapter 2

A Timed Temporal Logic Framework for Modeling Real-Time DESs

A series of processes need a description for modeling purposes, which leads to so called declarative models. These models are represented practically by predicates connected in rule-type formulas. Examples of such systems can be found in communication networks, computer programs, operating systems, manufacturing processes, etc. Being driven by events, which fire a rule or a set of rules in their models, these systems are called *Discrete Event Systems* (DES) or sometimes *Real-Time Systems* when the rules which describe them are involving the time as a variable, or event. Because of the rule nature of their models, one of the possible frameworks in which the analysis and modelling of such systems was done is related to temporal logic. In the next section a model for temporal logic will be presented and its connection to the real-time domain will be discussed.

2.1 Introduction

Temporal logic has its origins in philosophy. Studying the possibility of formalizing time framed statements philosophers [55] introduced special temporal operators such as $\Box$ (henceforth) and $\Diamond$ (eventually), for the analysis of temporal connectives in languages. As a valuable tool for analyzing the topology of time, various types of semantics can be given to the temporal logic operators.

Temporal logic was used in computer science for various purposes: Bruce [10] presented a formal model of temporal references in a natural language; Kahn and Gorry [30] introduced a time manager to handle temporal matters in the problem-solving routines; McDermott proposed a temporal logic for reasoning about processes and plans [40]; Allen elaborated what he calls a theory of time [3] as a part of a foundation for the sorts of temporal reasoning required in the AI applications; T. Dean et al. in [14, 13], used it in temporal database management,etc. In recent years, temporal logic has found applications in the areas of software verifications and knowledge based systems [23, 28]. For example, to specify program behavior, the structure of states is the key concept that makes temporal logic suitable for program specifications. A formula, containing temporal logic operators, is interpreted over a structure of states. In programming languages, a structure of states, which represents the computations executed by a program is used to interpret formulas. The latter represents temporal logic formulas that have been formalizing assertions expressing how the state of a program changes with time [38, 39, 41, 12].

Temporal logic, allowing a high level description of state transitions, was also used for the formal approach of control problems of DESs [20, 61].In the sequel of this chapter, we will introduce a temporal logic framework to the control problem of Real-Time DESs [45]. It is a discrete time temporal logic, and its language and proof system are adapted from those used in the verification of concurrent programs by Manna and Pnueli [38, 39] and enriched with real-time Peano type of axioms [46, 2, 6]. In Section 2.2, we describe the syntax and semantics of the temporal logic language. In Section 2.3, we present a system

of proof rules, and then its soundness and completeness in Section 2.4. In Section 2.5, we apply this framework to the specifications and verification for the control problem of a DES. Following that, we talk about the extension of temporal logic for real-time systems. Finally in Section 2.7, we discuss the advantages and disadvantages of this framework.

2.2 The Language, Its Syntax and Semantics

A few symbols of elementary set theory are used in the language. In what follows, we consider the axiomatic (Gödel's axiomatic) of the set theory as known. We denote a set by a bold or capital letter such as S or Γ, or embracing the members of the set with curly braces such as $\{w\}$, $\{w_1, w_2\}$. Also, we use the ideas of set-membership $\in$, of set union $\cup$, and of set intersection $\cap$.

The symbols of the language include individual constant symbols; individual variable symbols; definitional identity $=$, function letters, predicate letters, logical connectives: $\neg$ (the negation symbol) and $\vee$ (the disjunction connective), and temporal operators: $\bigcirc$ (the next operator), $\Box$ (the henceforth operator), and $\mathcal{U}$ (the until operator).

There are different time points which may yield different truth values of formulas in the language. We assume that the set of time points is infinite, discrete, and linearly ordered by the usual precedence relations $<$ and $\leq$. Hence we use the positive natural numbers as a set of time points.

The model we are going to work with is a many-sorted one $\mathcal{M} = (\mathbf{T}, \mathbf{D}, p^{\mathcal{M}}, f^{\mathcal{M}})$ where: p and $f \in \mathbf{F}^*$ are flexible or rigid predicates or functions, respectively $\mathbf{F}^* = \mathbf{F}_1^* \cup \mathbf{F}_2^*$, $\mathbf{F}_1^*$, $\mathbf{F}_2^*$ being the sets of all subsets of $\mathbf{F}_1$, respectively $\mathbf{F}_2$, the set of all flexible, and rigid logical formulas respectively; $\mathbf{T} = (T, 0, suc, \leq)$ and $\mathbf{D} = (D, p^{\mathbf{D}}, f^{\mathbf{D}})$ is the data structure [49]. We consider that $\mathbf{F}_1^* \cap \mathbf{F}_2^* = \emptyset$, and that $T \cap D = \emptyset$.

For each domain, following Enderton [16], we assume that we have a non-empty set I, whose members are called sorts. There is a countable number of constant symbols,

and also of variable symbols, of each sort i. For each $n > 0$ and each $(n+1)$-tuple of sorts $(i_1, i_2, ..., i_{n+1})$, there is a countable set of $n-$ary function letters said to be of sort $(i_1, i_2, ..., i_{n+1})$. Also, for each $n > 0$ and each n-tuple of sorts $(i_1, i_2, ..., i_n)$, there is a countable set of $n-$ary predicate letters said to be of sort $(i_1, i_2, ..., i_n)$.

We restrict the functions to range over the temporal subdomain, calling these flexible functions, or over the non-temporal subdomain, calling these rigid functions. Therefore the sort of a function will be uniquely determined by its range. We take the constant C to be a 0-ary function symbol. Hence, the constants are sorted as well. In addition, we will call predicates that take only temporal arguments temporal predicates, and predicates that take only non-temporal arguments non-temporal predicates [6].

Terms and wffs are defined in the standard fashion, with the only restriction being that arguments of the correct sort must be given for each function and predicate.

For any sort i, the terms of sort i are defined as those of a standard first order logic. In addition:

- the expression $(\bigcirc t)$, for any term t of sort i, is a term.

Definition 2.2.1: The well-formed formulas (or simply, formulas) of the language are defined as follows:

(i): for any terms $t_1, \cdots, t_n$, any predicate symbol P, $P(t_1, \cdots, t_n)$ is a formula;

(ii): for any formula w, $(\neg w)$, $(\bigcirc w)$, and $(\Box w)$ are formulas;

(iii): for any formulas v and w, $(v \vee w)$ and $(v \mathcal{U} w)$ are formulas.

Informally, $(\bigcirc w)$ means that w will be true at the next time instant; $(\Box w)$ has the English paraphrase: w will be true henceforth; and $(v \mathcal{U} w)$ has the intuitive interpretation: v will be true until w is (and w will indeed eventually be true).

In the language, a state, an interpretation, and a structure are defined as follows:

Definition 2.2.2: Let D_i be the domain containing x, y, z, $\cdots$ for every sort i. A state is an assignment of an element of D_i to each variable symbol of sort i. An interpretation M specifies a domain D_i for each sort i, and assigns to each constant symbol of the sort i an element of D_i, to each function letter of sort $(i_1, i_2, ..., i_{n+1})$ a function $f : D_{i_1} \times D_{i_2} \times \cdots \times D_{i_n} \rightarrow D_{i_{n+1}}$ and to each predicate letter of sort $(i_1, i_2, ..., i_n)$ a relation $P \subset D_{i_1} \times D_{i_2} \times \cdots \times D_{i_n}$.

Definition 2.2.3: A structure is defined by (M, σ) where

- M is an interpretation; and
- σ is an infinite sequence of states $s_0, s_1, s_2, ...$

The formulas are evaluated with respect to the structure (M, σ). For the structure (M, σ) with sequence $\sigma = s_0 s_1 s_2 \cdots$, and for any integer $k \geq 0$, $M^{(k)}$ represents the structure $(M, \sigma^{(k)})$ with sequence $\sigma^{(k)} = s_k s_{k+1} \cdots$, and $M^{(0)} = M$. The following notations are used:

- $M[t]$ represents the value assigned to the term t by M;
- $M[C]$ represents the value assigned to any global constant symbol C by M;
- $M[x]$ represents the value assigned to any local variable symbol x under the initial state of σ by M;
- f^M represents the function assigned to the function letter f by M;
- P^M represents the relation assigned to the predicate letter P by M.

Proposition 2.2.1: For any term t, $M[(\bigcirc t)] = M^{(1)}[t]$.

Intuitively, the states of the sequence of a structure represent the state of affairs at successive instants of time. So, the global constant symbols, predicate letters and function letters have interpretations that are independent of time, while the local variable symbols represent quantities that change with time.

Definition 2.2.4: A state formula is any well-formed first order formula which is evaluated with its truth value depending only on the first state of the path at which it is evaluated. A temporal formula is a formula constructed from state formulas to which some temporal operators are applied.

Definition 2.2.5: A formula w is said to be satisfied by a structure (M, σ), written by $\models_M w$, if it is defined as follows:

(i). for any formula of the form $P(t_1, t_2, ..., t_n)$ where P is a predicate letter and $t_1, t_2, ..., t_n$ are terms, $\models_M P(t_1, t_2, ..., t_n)$ if and only if

$$(M[t_1], M[t_2], ..., M[t_n]) \in P^M$$

(ii). $\models_M (\neg w)$ if and only if it is not the case that $\models_M w$;

(iii). $\models_M (v \vee w)$ if and only if either $\models_M v$ or $\models_M w$;

(iv). $\models_M (\bigcirc w)$ if and only if $\models_{M(1)} w$;

(v). $\models_M (\Box w)$ if and only if for all $i \geq 0$, $\models_{M(i)} w$;

(vi). $\models_M (v \mathcal{U} w)$ if and only if there exists an $n \geq 0$ such that $\models_{M(n)} w$ and for all k, $0 \leq k < n$, $\models_{M(k)} v$.

(ii) in Definition 2.2.5 is the definition of that the negation of a formula w is true; and it is defined as not the case of that w is true. It is equivalent to the Closed World Assumption (CWA)[28] which says what we do not know can not be true. In a sense of the CWA, we assume that the state descriptions completely characterize the positive facts of a model. For example, we use $fire(GUN)$ to represent the fact of that the trigger of the gun is pulled. If we say that the trigger of the gun is not pulled at time 1, then this means it is not the case of $fire(GUN)$, that is $\neg fire(GUN)$ is true, at time 1. So if $fire(GUN)$ is not given in the description, then it implies that $\neg fire(GUN)$ is true. Another example, if $W = P$ is not in the formula, then this means that it is not the case of $W = P$, that is, $\neg W = P$ is true (we usually write this as $W \neq P$).

The following are some convenient abbreviations to be used in the sequel:

(i). conditional implication: $v \Rightarrow w$ abbreviates $(\neg v \vee w)$;

(ii). conjunction connective: $v \wedge w$ abbreviates $(\neg(\neg v \vee \neg w))$;

(iii). biconditional implication: $v \Leftrightarrow w$ abbreviates

$$(v \Rightarrow w) \wedge (w \Rightarrow v);$$

(iv). eventually operator: $(\Diamond w)$ abbreviates $(\neg(\Box(\neg w)))$ and has the English paraphrase: w will eventually be true;

(v). precedes operator: $(v \mathcal{P} w)$ abbreviates $(\neg((\neg v)\mathcal{U} w))$ and has the intuitive interpretation: v will become true before w does.

Definition 2.2.6: A formula w is said to be valid, written by $\models w$, if $\models_M w$ for all M.

2.3 The Proof System

The proof system has three parts: the general part, the formal mathematical reasoning, and the description of events. The general part deals only with the formal temporal and logical reasoning.

2.3.1 The general part

Axiom schemata

If a formula w has the form of one of the following schemata, then w and $\Box w$ are axioms:

(A0) w, where w is an instance of a propositional tautology;

(A1) $\bigcirc(w_1 \Rightarrow w_2) \Rightarrow (\bigcirc w_1 \Rightarrow \bigcirc w_2)$;

(A2) $\neg \bigcirc w \Leftrightarrow \bigcirc \neg w$;

(A3) $\Box(w_1 \Rightarrow w_2) \Rightarrow (\Box w_1 \Rightarrow \Box w_2)$;

(A4) $w_1 \mathcal{U} w_2 \Rightarrow \Diamond w_2$;

(A5) $\Box w \Rightarrow w \wedge \bigcirc \Box w \wedge \bigcirc w$;

(A6) $w_1 \mathcal{U} w_2 \Leftrightarrow [w_2 \vee (w_1 \wedge \bigcirc(w_1 \mathcal{U} w_2))]$;

(A7) $\Box(w \Rightarrow \bigcirc w) \Rightarrow (w \Rightarrow \Box w)$;

(A8) $t = t$, for any term t;

(A9) $t_1 = t_2 \Rightarrow (w(t_1, t_1) \Leftrightarrow w(t_1, t_2))$, 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;

(A10) for any $n-$ary predicate letter P and terms $t_1, t_2, ..., t_n$

$$\bigcirc P(t_1, t_2, ..., t_n) \Leftrightarrow P((\bigcirc t_1), (\bigcirc t_2), ..., (\bigcirc t_n))$$

(A11) $\bigcirc f(t_1, t_2, ..., t_n) = f((\bigcirc t_1), (\bigcirc t_2), ..., (\bigcirc t_n))$ for any $n-$ary function letter f, terms $t_1, t_2, ..., t_n$ and equality symbol $=$; and

$C = (\bigcirc C)$, for any constant symbol C and equality symbol $=$.

Inference rule:

Modus ponens: for any formulas w_1 and w_2

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

Here we use the notation

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

to mean that one may infer the formula w_{n+1} from the formulas $w_1, w_2, ..., w_n$. Also the notation $\vdash w$ means that the formula w is a theorem of our proof system, and $\Gamma \vdash w$ means that w can be deduced from the set Γ of formulas.

Generalization rule:

If $\{w_1, w_2, ..., w_n\} \vdash w_{n+1}$, then $\{\Box w_1, \Box w_2, ..., \Box w_n\} \vdash \Box w_{n+1}$.

Propositional reasoning (PR):

If

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

is an instance of a tautology, then

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

Deduction theorem:

$\Gamma \cup \{w_1\} \vdash w_2$ if and only if $\Gamma \vdash w_1 \Rightarrow w_2$ for any set of formulas $\Gamma \cup \{w_1, w_2\}$.

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:

$$\Box(w_1 \wedge w_3 \Rightarrow \bigcirc w_2)$$

$$w_1 \Rightarrow \Diamond w_3$$

$$\frac{\Box(w_1 \Rightarrow (\bigcirc w_2 \vee \bigcirc w_1))}{w_1 \Rightarrow (w_1 \mathcal{U} w_2)}$$

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:

$$\vdash \Box[\bigcirc w(y_1, y_2, ..., y_n) \Leftrightarrow w((\bigcirc y_1), (\bigcirc y_2), ..., (\bigcirc y_n))]$$

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

2.3.2 Formal mathematical reasoning

Manna and Puneli [38] include in their proof system a 'domain part' to do formal mathematical reasoning about the domains of their interpretation. Since the main interest for DESs here is temporal logic reasoning, all of those formulas that are true under intended evaluation are axioms:

(A12) if w is a formula that is true under the obviously intended evaluation, then w and $\Box w$ are axioms.

We call such axioms domain axioms. In order to further simplify our formal mathematical reasoning, we state the following derived rule:

Mathematical reasoning (MR):

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

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

2.3.3 The description of events

One of the sorts of the language is used to represent events which include various transitions such as actions and operations. The global constant symbols of this sort are called *event symbols* and the symbol δ is included among the local variables of that sort. Thus, the formula $\delta = \alpha$ means that the event represented by the event symbol α is about to occur.

The sequence of a structure is infinite, but it may be that a system eventually stops making state transitions because a goal has been achieved and no further action is required,

or because a deadlock has occurred and no further transitions are possible. In order that such situations can be modelled, the symbol ϵ is included among the event symbols and stands for the null event. This event leaves unchanged the values of all local variable symbols; and it gives yet another axiom schema of the proof system as follows:

(A13) $\Box[\delta = \epsilon \Rightarrow (\bigcirc x) = x]$ where x is any local variable symbol.

2.3.4 The description of time

Originally the temporal logic operators were introduced in order to abstract from time. In other words, the explicit presence of the time variable in temporal logic formulas was removed. However, for real-time systems, there is a need for explicitly marking the state evolution in time.

There are many approaches in which the time-marking of the states is looked for. Among them are: real-time automata theory [4, 5], real-time process algebra [7], mu-calculus in real-time [15], timed transition systems [25], timed transition models [47] to quote just a few.

Real-time systems are characterized by quantitative timing properties relating occurrences of events, for example, the exact time between events, and the maximal or minimal time between events. In [53], Pnueli and Harel give a brief account of some attempts to use temporal logic for the specification of real-time systems. The computational model used is a timed interleaving model where enabled transitions have associated lower and upper bounds within which they must be taken. It considers two possible extensions of temporal logic to deal with real-time. The first adds a global clock as an explicit variable to which the specification may refer. The second approach introduces quantitative temporal operators. For specifying synchronous systems, it recommends the use of a discrete time domain such as the natural numbers, and for asynchronous systems, a dense time domain such as the rational numbers.

One of the methods using the first approach is that of Ostroff [45, 44] for the control of real-time DESs. It introduces a distinguished variable t representing the clock. A typical formula of his logic is the following:

$$\varphi \wedge t = T - \Diamond(\psi \wedge t \leq T + 5) \tag{2.6.1}$$

where T is a global variable [45]. (2.6.1) says that if φ is true now and the clock reads T ticks, then within $T + 5$ clock ticks ψ must become true. Thus, once φ becomes true, ψ must become true no more than 5 ticks later. The explicit clock variable allows the easy reference to times when specifying the properties of the real-time. However, the explicit clock variable is against the original philosophy of temporal logic to abstract from time as much as possible.

An example using the second approach is that of Koymans [32]. It extends temporal logic with metric operators from their qualitative versions. For example, the semantics of (2.6.1) can be written equivalently by the metric temporal logic as follows:

$$\varphi - \Diamond_{\leq 5}\psi \tag{2.6.2}$$

The metric operators have been applied to the formal specifications of real-time systems. However, there is no proof system in the metric temporal logic so that it lacks means of verification.

In what follows we will introduce the basics of the timed-temporal logic which we will use in the sequel.

The time set was already introduced above as $\mathbf{T} = (T, 0, suc, \leq)$. In other words, it is made out of a set of natural numbers, which has endowed with a neutral element 0. It has a generator called *suc*, and is ordered by $\leq$, or formally [8]:

(P1) $0 \in Nb$

(P2) $s : Nb - Nb$;

(P3) $(\forall x)(\forall y)(\forall z)((z = s(x) \wedge z = s(y)) \Rightarrow x = y)$;

(P4) $(\forall x)\neg(0 = s(x))$;

(P5) $(\forall)((\pi(0)\wedge(\forall x)(\forall y)((y = s(x)\wedge\pi(x)) \Rightarrow \pi(y))) \Rightarrow (\forall z)\pi(z))$ *where* $y, z \notin var(\pi(x))$;

The axiom (P5) it is usually called the transfinite induction principle, which has the following temporal logic expression:

$$A(n) \Rightarrow \Diamond(B \vee (\exists m)(m < n \wedge A(m)))$$

[33], where $B_1 \Rightarrow \Diamond B_2$, $B_2 \Rightarrow \Diamond B_3$, ..., $B_{k-1} \Rightarrow \Diamond B_k$ where $k \geq 2$ is a set of formulas which we can prove, and $A_{k-1} \stackrel{\text{def}}{=} B_1$, $A_{k-2} \stackrel{\text{def}}{=} B_2$, ..., $A_1 \stackrel{\text{def}}{=} B_{k-1}$ and $B = B_k$. It is shown that the above axioms determine N to isomorphism[33]. The rational set can now be generated from N by multiplying the elements of N by rational ratios of type $\frac{m}{n}$ where $m, n \in Nb$ are prime natural numbers **Q**.

To introduce now a time variable it suffices to consider a generator of rational numbers. It was argued by Bacchus et al. that the rational number set is sufficient for correctly characterizing the problems we might encounter in real life. The rational number generation process is then figured as: "$\kappa = const * n$" where $\kappa, const \in \mathbf{Q}$ and $n \in \mathbf{N}$. It is worth noting that there might be more than one time variable, each system or part of a system might have its own time variable. A common time variable called the "global time" of the system will be used to be able to synchronize correctly time events and correspondingly state variables. The introduction of a global time variable requires the statement of the following axioms:

(A14) $\Box[\delta_{GlobClock} = START \wedge IdleGlobClock(x) \Rightarrow \bigcirc ADD1(\kappa)]$ where ADD1(x) stands for that x is increased by 1;

(A15) $\Box[\neg EQ(\kappa, 0) \Rightarrow (\kappa = \bigcirc ADD1(\kappa))]$

(A16) $\Box[IdleGlobClock = EQ(\kappa, 0)]$

The above two axioms help start a global clock (A14) and make the ticks of the clock increase (A15).

For any other clock in the system the ticks are coordinated by the global clock through rules that are written:

$$\Box[\delta_{clock(x)} = START \wedge IdleClock(x)(\kappa(x) = \bigcirc ADD1(\kappa(x)))]$$

where $\kappa(x) = \sigma_T \kappa$, $\sigma_T \in \mathbf{Q}$, and x is any local variable symbol. Their idle state is then defined by:

$$\Box[IdleGlobClock(x) = EQ(\kappa, T_{clock(x)})]$$

where $T_{clock(x)} \in \mathbf{T}$.

Due to the introduction of the clock ticks, some other process axioms have to be introduced. These are:

(A17) $\Box[s(x) \wedge e(x) \wedge \kappa(x) = \sigma_T n \Rightarrow \mathcal{U}(\bigcirc s'(x) \wedge (T_a \leq \kappa \leq T_b)]$ where x is any local variable symbol, σ_T is an arbitrary constant which fixes the speed of the local clock, T_a and T_b are time bounds where T_a, $T_b \in T$.

The last axiom allows also for writing rules for systems which have to specify an action taking place at a specific time moment.

2.4 Soundness and Completeness

As mentioned in Subsection 2.3.1, the notation $\vdash w$ means that the formula w is a theorem of the proof system, and $\Gamma \vdash w$ means that w can be deduced from the set Γ of formulas. The notation $\Gamma \models w$ means w is a consequence of Γ; and it is defined by using Definition 2.2.6 as follows: $\Gamma \models w$ if $\models v$ for every $v \in \Gamma$ implies $\models w$.

The proof system of the temporal logic without considering the Peano's axioms is sound, strongly sound, and complete; but not strongly complete as given below.

Definition 2.4.1: A proof system is sound iff $\vdash w$ implies $\models w$ for all w; and it is strongly sound iff $\Gamma \vdash w$ implies $\Gamma \models w$ for any Γ and w.

Theorem 2.4.1: The proof system given in Section 2.3 is sound and strongly sound [61].

Definition 2.4.2: A proof system is complete iff $\models w$ implies $\vdash w$ for all w; and it is strongly complete iff $\Gamma \models w$ implies $\Gamma \vdash w$ for any Γ and w.

Theorem 2.4.2: The proof system given in Section 2.3 is complete but not strongly complete [61].

However, by adding the Peano's axioms the proof system will inherit the deficiencies of any arithmetical system known so far. The following theorem gives the fundamentals for the above statement.

Theorem 2.4.3: Let $\mathcal{T} \supseteq \mathrm{N}_0$ be an effectively axiomatized theory in which N is a model. Then $\mathcal{T}$ is incomplete [8].

The result mentioned above stems from the fact that an element $q \in \mathcal{L}(\mathcal{T})$ can be constructed such that neither q nor $\neg q$ is a theorem of $\mathcal{T}$. The proof of this theorem shows that no effective axiomatization of N can lead to a complete theory.

Abadi, however, in [2] proposes alternate notions for completeness, translating temporal formulas into classical formulas with explicit time parameters. Following his developments one can state that the temporal logic enriched with Peano's Arithmetic is sound and complete as in the next two theorems.

Theorem 2.4.4 (Soundness): For every formula w $\vdash_{T_1} w \Rightarrow \vdash_O P(w)$ and $\vdash_{T_2} w \Rightarrow \vdash_P P(w)$, where O is any temporal logic (classical) system, T_1, and T_2 are temporal logics in which the time set T ordered or endowed with the induction principle respectively, and P denotes the Peano's axiomatic.

Theorem 2.4.4 (Completeness):

(1) For every formula $w \vdash_O P(w) \Rightarrow \vdash_{T_1} w$ if w is arithmetical in $\models_{T1}$ and

(2) For every formula $w \vdash_P P(w) \Rightarrow \vdash_{T_2} w$.

Despite Abadi's work, Theorem 2.4.4. is shown to be inaccurate by Sain in [49].

However, because completeness means that some formulas hold in every intended model, but cannot be proved, because our method of synthesis automatically does the verification, and mainly because (ii) of Definition 2.2.5., we are merely inclined towards Abadi's work.

This completes the descriptions of the language and proof system of the temporal logic framework which will be the fundamental base of the research developed towards the design and implementation of the temporal logic language of our environment.

For a DES, if a structure satisfies the given plant specification, then it represents a physically possible trajectory of the system. If, in addition, it satisfies the controller specification, then it represents a possible trajectory of the closed-loop system. Hence, the temporal logic framework provides a way to verify the properties in the analysis process of timed deterministic DESs.

A simple example will be described next illustrating this approach.

2.5 An Example of Real-Time Systems

The alternating bit protocol (ABP) has been taken to show the usefulness of the timed temporal logic (TTL) described above. We will refer to the same example described by Lin in [36] as means of comparison.

The ABP is used to coordinate the flow of messages between two processes [24] , a *Sender* process on one node and a *Receiver* process on another, in a distributed network. The *Sender* transmits a packet and it will not transmit another packet until it receives an acknowledgement that the packet was received correctly by the *Receiver*. Similar to the notation used in [36], the states of the sender are either 0, 1, or n. The state "0" represents

that the sender is waiting for ACK0. It is "1" when sender is waiting for ACK1, and it is "n" when the sender is not waiting for an ACK. The state of the receiver is "0" if it is waiting for packet number 0, and it is "1" if it is waiting for packet number 1. The state of the protocol is $x = (s, r)$ where s is the state of the sender and r is the state of the receiver. Hence we can define the protocol having six states: (0,0), (n,0), (0,1), (n,1), (1,1), (1,0). And the events would be as follow:

- packet 0 is lost ($ep0lo(.)$);
- packet 0 is received ($ep0rc(.)$);
- packet 1 is lost ($ep1lo(.)$);
- packet 1 is received ($ep1rc(.)$);
- acknowledgement $ACK0$ is lost ($ea0lo(.)$);
- acknowledgement $ACK0$ is received ($ea0rc(.)$);
- acknowledgement $ACK1$ is lost ($ea1lo(.)$);
- acknowledgement $ACK1$ is received ($ea1rc(.)$);
- timeout and packet 0 is received or lost ($ep0to(.)$);
- timeout and packet 1 is received or lost ($ep1to(.)$);

Real-time systems are characterized by quantitative timing properties relating occurrences of events. In [36] the temporal logic specification defines only invariants, eventualities, and order constraints on the sequences of the states. With the TTLMs developed here, the specifications are defined incorporating their real-time properties.

The global clock is initialized through the following rule:

$$\Box[\delta_{GlobClock} = START \wedge IdleGlobClock(x) \Rightarrow \bigcirc ADD1(\kappa)],$$

which is incremented by:

$$\Box[\neg EQ(\kappa, 0) \Rightarrow (\kappa = \bigcirc ADD1(\kappa))]$$

Two local clocks ($\delta_{clock(x)}$) coordinated with the global clock have been defined; one assigned to the sender and another to the receiver. They might be at any time different from one another. This local clock is defined by:

$$\Box[\delta_{clock(x)} = START \wedge IdleClock(x)(\kappa(x) = \bigcirc ADD1(\kappa(x)))]$$

Let the local variables s and r represent the states of the protocol using the predicates and events described above. The rules of the system are written following axiom (A17). They allow to specify the actions that take place at specific moments of time or time interval. The set of TTL formulas of the specifications for the AB protocol is given as follows:

Dynamics

$$\Box[(ep0rc(r) \wedge n(s) \wedge 0(r) \wedge t = \kappa) \wedge \mathcal{U}(0 \leq \kappa \leq 6) \Rightarrow 0(\bigcirc s) \wedge 1(\bigcirc r)] \qquad (2.5.P1)$$

$$\Box[(ep0lo(s) \wedge n(s) \wedge t = \kappa) \wedge \mathcal{U}(1 \leq \kappa \leq 3) \Rightarrow 0(\bigcirc s)] \qquad (2.5.P2)$$

$$\Box[(ea0rc(s) \wedge 0(s) \wedge t = \kappa) \wedge \mathcal{U}(4 \leq \kappa \leq 6) \Rightarrow n(\bigcirc s)] \qquad (2.5.P3)$$

$$\Box[(ea0lo(r) \wedge 1(r) \wedge t = \kappa) \wedge \mathcal{U}(3 \leq \kappa \leq 6) \Rightarrow 1(\bigcirc r)] \qquad (2.5.P4)$$

$$\Box[(ep0to(s) \wedge 0(s) \wedge t = \kappa) \wedge \mathcal{U}(6 \leq \kappa \leq \infty) \Rightarrow 0(\bigcirc s)] \qquad (2.5.P5)$$

$$\Box[(ep0rc(r) \wedge 0(r) \wedge t = \kappa) \wedge \mathcal{U}(2 \leq \kappa \leq 6) \Rightarrow 1(\bigcirc r)] \qquad (2.5.P6)$$

$$\Box[(ep0lo(s) \wedge 0(s) \wedge t = \kappa) \wedge \mathcal{U}(2 \leq \kappa \leq 6) \Rightarrow 0(\bigcirc s)] \qquad (2.5.P7)$$

$$\Box[(ep1lo(s) \wedge n(s) \wedge t = \kappa) \wedge \mathcal{U}(1 \leq \kappa \leq 6) \Rightarrow 1(\bigcirc s)] \qquad (2.5.P8)$$

$$\Box[(ep1rc(r) \wedge 1(r) \wedge t = \kappa) \wedge \mathcal{U}(2 \leq \kappa \leq 6) \Rightarrow 0(\bigcirc r)] \qquad (2.5.P9)$$

$$\Box[(ep1lo(s) \wedge 1(s) \wedge t = \kappa) \wedge \mathcal{U}(2 \leq \kappa \leq 6) \Rightarrow 1(\bigcirc s)] \qquad (2.5.P10)$$

$$\Box[(ea1lo(r) \wedge 0(r) \wedge t = \kappa) \wedge \mathcal{U}(3 \leq \kappa \leq 6) \Rightarrow 0(\bigcirc r)] \qquad (2.5.P11)$$

$$\Box[(epIto(s) \wedge 0(s) \wedge t = \kappa) \wedge \mathcal{U}(6 \leq \kappa \leq \infty) \Rightarrow 0(\bigcirc s)] \qquad (2.5.P12)$$

$$\Box[(ca1rc(s) \wedge 1(s) \wedge t = \kappa) \wedge \mathcal{U}(4 \leq \kappa \leq 6) \Rightarrow n(\bigcirc s)] \qquad (2.5.P13)$$

Formula 2.5.P2 describes the events in which packet 0 has been lost; this event can occur when the state of the sender is in the initial condition, that is, it is not expecting an ACK signal. The temporal operator sets the timing constraints and makes the rule false if the global clock is out of the time bounds, in other words, the rule is true until the time constraint is met. The current state of the sender changes from $n(s)$ to $0(s)$, meanwhile, the state of the receiver remains the same.

Similarly to rule 2.5.P2, 2.5.P3 represents the state transition during which the sender changes the state from being waiting for ACK0 to idle. For this transition the event must be generated while the clock is at any time between the lower and upper bounds, 4 and 6 respectively, and it is true until the time constraint is met. The remaining formulas describe the effects of event occurrences, the validity of these formulas are constrained by the temporal operators along with the values of the clock.

Figure 2.1 shows how the time bounds affect the simulation process. When the system was at state "SENDER0 RECEIVER1" the next event led the system to receive the acknowledge for packet 0, but it was received out of the upper time bound indicating that the packet 0 was timed out, and has to be resent.

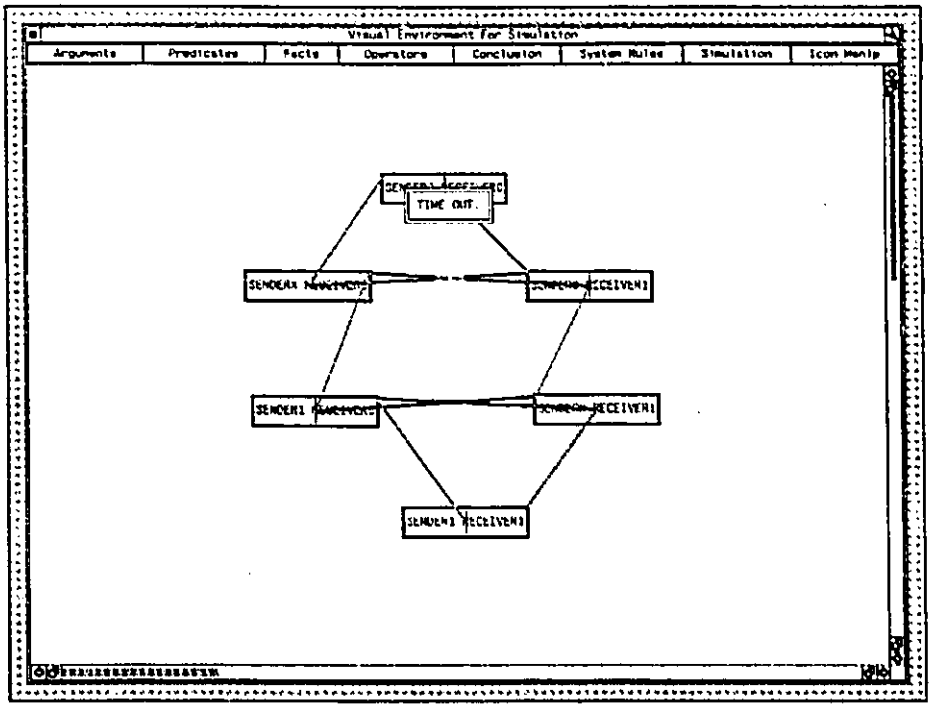

Figure 2.1: Real-time Simulation Results of the ABP Example.

2.6 Discussion

In this chapter, we have introduced the research background as a starting point of this thesis. As a general background, we have briefly introduced the development of applications of temporal logic to the AI reasoning, database management, software verification and high-level specifications.

We have introduced a discrete linear-time temporal logic framework and added to it Peano's axioms for Arithmetic. The system of axioms is close to the construction of the timed temporal logic of Abadi. The framework is a starting point of this thesis.

For the real-time applications, we have talked about the development on the real-time aspects of temporal logic. We have built our own abstract system following Abadi [2], Sain

[49], and Bacchus [6].

The temporal logic framework presented in Sections 2.2-2.5 uses a linear-time temporal logic adapted from that of Manna and Pnueli [38] enriched with N as Abadi's T_2 temporal logic; so that a proof system is available based on their work although it is slightly different from their proof system. In this framework a sort of the temporal logic is set aside for the events of the system. This allows for easy references to events in the specifications. On the time side we have introduced a set of explicit time rules for generating general and local clocks. This allows for easy references to clock ticks and states labeled by the ticks. Using the temporal logic framework presented in Sections 2.2-2.5, we were able to compare with an example presented in [36] in which the real-time aspect is not considered.

A major disadvantage of this approach is that the full plant-controller system must be available before verification can proceed. To overcome this disadvantage, we will develop the method to do an early verification, *i.e.* to verify the specification of the DES's plant before or without designing a controller.

The framework introduced here can be improved to be more expressive to deal with the probabilistic features of DESs.

Chapter 3

A Graphical Programming Environment (GPE) For A Timed Temporal Logic Language

As window systems become popular, there is a growing need for Graphical User Interfaces (GUIs) that allow users to manipulate objects on the screen directly. This chapter presents a timed temporal logic language and the general architecture for implementing a visual programming environment for this temporal logic language. The temporal operators used in the temporal logic language are briefly discussed. A description of the alphabet and grammar of the visual language will be given in Section 3.7. And, in Section 3.6 a counter or clock as means of timing constraints within the temporal logic models to verify the system properties and dynamic behavior is introduced.

3.1 Introduction

In Chapter 2 we showed that temporal logic provides the capabilities for reasoning about time behaviors of discrete systems, such discrete time based approaches are inadequate when certain real-time properties are to be precisely specified, especially in the case

of real-time concurrent systems. We introduced the temporal logic framework which is a linear-time temporal logic for real-time DESs that incorporates a global clock. Chapter 2 also introduced the basis of the timed-temporal logic which considers a common time variable called the "global time" of the system used to synchronize correctly time events and corresponding state variables. However, there might be more than one time variable, each system or part of the system might have its own time variable.

The objective of the software development process is to produce a system which meets the user's requirements. There have been a lot of developments towards designing user-friendly graphical interfaces. GUIs are tools very attractive from the view point of flexibility since the user is responsible for defining system models using graphical primitives. Use of multiple windows, pop-up menus, and mouse operations provide a very friendly environment for the user.

Graphical languages give users the ability to perceive problems in a format that is closer to their mental representation of such problems, allowing data to be manipulated closer to the way they are manipulated in real world.

In a visual programming environment the goal to achieve is to create a set of graphical icons such that the user is allowed to design a solution to the problem at a very high level by interconnecting such icons and then directly execute the resulting iconic diagram. For the case of DESs in a temporal logic framework the solutions to the problems will be to write the temporal logic models that reflect the behavior of the system.

One important goal in visual programming is to present to the user a visual syntax that matches the user's conceptual model in the construct of the language described in chapter 2, such syntax should be understandable to both the machine and the user. The temporal logic model components have to be represented in such a way that in the process of building a TTLM, the connection of the iconic components can be perceived by intuition by the user.

3.2 Interactive Graphical User Interface

The GPE designed throughout this research work is used to create TTLMs for DESs. A GUI should provide the linkage between the user and the rest of the system components and should allow the user to retrieve information that has been previously stored which might be subsequently modified as well as reused.

A good graphical user interface methodology is expected to meet certain criteria when used in designing visual environments:

- It should facilitate easy use of the environment,
- The graphical programming scheme should be straightforward,
- It should minimize programming errors, and
- It should serve as a good communication medium for people.

A major paradigm in user-interface design is direct manipulation, which enables the user to specify a model by interaction on a graphical representation, as an alternative to the abstract and error-prone textual input technique. Although direct manipulation offers an intuitive interface style, it does not offer any facilities with respect to the model representation. A high-level model representation capable of generating alternative configurations without the need to rebuild the model after a change is essential.

One way to obtain high-level model representation is by using a procedural language to describe the temporal logic models. The TTLMs can be broken into smaller lexical objects, say, arguments, predicates, temporal operators, logical operators, etc., every one of which has a representation in the model. A set of rules or constraints establishes the relation that must be maintained among the lexical objects. For instance, in the graphical representation an argument object will never be connected to a logic operator.

When aimed toward simulation of DESs in a temporal logic framework, the GUI should

allow the user to participate in all the phases of the simulation process: from building the TTLMs until verifying the system behavior. These phases can be summarized as follows:

- Graphical creation of a database for facts of the rule base system.
- Graphical creation of temporal logic models of the system.
- Intelligent advisory to guide the user in accomplishing correct specifications.
- Graphical creation of temporal logic models for the system controller following some recommendations.
- User interaction with the system providing mechanisms to stop the simulation process, make changes, etc.
- Visualization of simulation model by means of animated changes in the states of a simulation model.
- Automatic error indicators (messages and prompts on the screen).

3.3 General Architecture

The general architecture of the visual programming environment is divided into three main blocks: the Graphical User interface (GUI), the Inference Engine, and the Visual Simulation. All these three blocks have their own classes that come into interaction through the main class called LogicalWS that stands for Logical Work Space (See Figure 3.1). A more detailed picture of the General Architecture of the VPE is depicted in Appendix E.

Temporal logic models are generated by interconnecting icons on the screen. Once the system knowledge is introduced in the database, the simulation block is called, however, before any simulation tasks take place the inference engine comes in to play its role, that is, parse the temporal models already written and reason about the system according to the knowledge. Then, the simulation block generates the set of states and corresponding events.

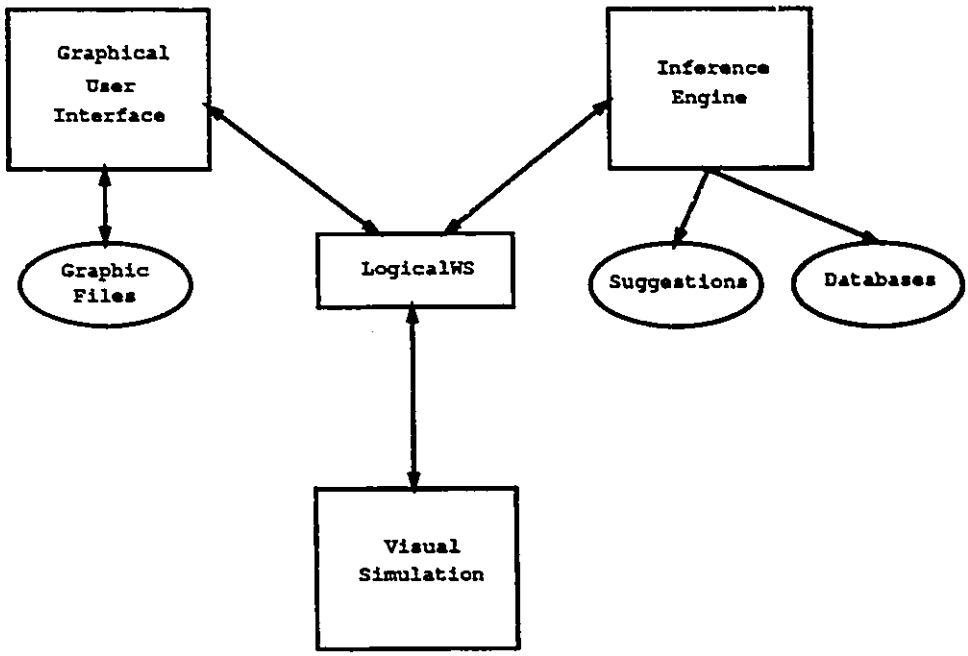

Figure 3.1: General Architecture.

Every single class interacts with all the others, as well as some interaction with input/output files where initial conditions, data previously saved, and results are stored. Data is read from and written to databases in order to be reused.

The main block of this interactive visual environment is represented by the class LogicalWS. It creates the menus, the physical workspace, and the logical environment on which all the actions of the simulation of systems take place.

The LogicalWS class creates a class object, logicalSpace, whose tasks are:

- Creates the graphical interface that will allow the user to build up applications using graphic symbols.
- Creates and initializes the working memory. The working memory comprises a series of ordered collection objects where the initial databases of library items are stored, which later might be, if needed, reused. These ordered collections are:
  - FactDataBase. This ordered collection stores the information of the facts which

represent the knowledge of the system.

- ◇ PredicateDataBase. This collection holds the predicate names that have been created for future reusability. It maintains the information about predicate name text strings and number of arguments any predicate can take.
- ◇ StatementDataBase. The information maintained in this collection is not used in the reasoning process, but it is used for reusing information instead when it comes to building rules since a statement, which consists of a predicate name and all the arguments and temporal operator attached, can be called from this database and immediately attached to the LogicalCls class objects.
- ◇ RuleDataBase. It holds the overall characteristics of the system through rules of the form IF(*premise*)THEN (*conclusion*). It plays a major role in the reasoning process. The content of this ordered collection is matched against the contents of the database of facts.
- ◇ BehaviorDataBase. It holds a set of rules which gives shape to the system behavior. These rules tell the system which behavior the system should have. And,
- ◇ ControlDataBase. This object is an ordered collection whose content is a set of rules with the information to model the controller which rules the general behavior of the controlled system.

Each of these databases works synchronized with its corresponding graphical file that contains both the logical information (text strings) and physical information (coordinates on the screen) for every class object previously stored or saved.

- Creates the menu bar from which the temporal logic model components are selected to build the applications.
- Permits the simulation tasks to run by simply mouse button press events, as well as interaction with the simulation process. An automatic simulation feature is added and it will be described later in this chapter.

- Initializes the instances of horizontal and vertical scrollbar classes to take advantage of the working space since the entire space is several times larger than the visible window.
- Performs maintenance tasks such as clear the working space, save an iconic diagram into a file, retrieve a file provided that the user enters the file name.

In creating temporal logic models, the components are selected from the main menu bar at the top most part, see Figure 5.1. By choosing any of the first five menu selections with the mouse, the programming environment will pop up a list of submenu options pertaining to that class. In order for the user to create or reuse a class object, he or she has to select the appropriate options from that particular submenu. The lexical components that can be created are of the following classes:

- ArgumentCls,
- PredicateCls,
- TemporalCls,
- LogicalCls, and
- Conclusion.

Any object that belongs to any of the above classes responds to its own menu of selections such as remove a parent or a child, save, quit, implode, etc.

3.4 Temporal Operators: Design Specifications and Strategies

Temporal logic models are constructed from predicate symbols, function symbols, individual constants, individual variables, the classical logical operators ($\neg$, $\vee$, $\wedge$), and the temporal operators ($\bigcirc$, $\Box$, $\Diamond$, and $\mathcal{U}$).

The temporal operators in temporal logic models are assumed to specify properties of the system states that could evolve during a time period as Ostroff stated in [15]. They represent a valuable tool for analyzing the topology of time, various types of syntax and semantics can be given to the temporal operators.

In DESs the temporal operators mentioned above make no reference to the time instant or time period when the states occur. A common time variable, a global clock, used for reasoning about real-time events is added to these temporal operators as a means of including timing constraints and real-time properties to the formulas.

Let Γ be a well-formed formula. A temporal expression w is defined as: w is true if Γ is true now (at present time), for instance $\Gamma(0)$ = true. While the realization of Γ at a given time assigns a truth value to that single point of time, the temporal expression w assigns a single truth value to the whole time interval. Its value cannot change in time. In other words, the truth value of temporal expressions depends on the corresponding time period. If for instance Γ is true only sometimes, then $\Diamond w$ is true and $\Box w$ is false. This single truth value covers the whole time period. The temporal operators $\bigcirc w$, $\Box w$, $\Diamond w$, and $\mathcal{U}w$ have only a single truth value which regards the whole time period implicitly but this value itself is not explicitly time-dependent and hence it cannot be changed in time. $\Box w$ is true or false; this refers to the whole time period.

In the visual programming environment, each temporal operator is a template with an iconic representation. It would be very nice to symbolize each temporal operator by means of their graphical representation; that is, $\bigcirc$, $\Box$, $\Diamond$, and $\mathcal{U}$; to represent the temporal operators NEXT, HENCEFORTH, EVENTUALLY, and UNTIL, respectively. This is not possible due to constraints of the software programming language used.

A temporal operator in the language defined in the previous chapter is a lexical component attached to a predicate. This temporal operator sets the behavior of the predicate in time along with the time bounds. In the design of the GPE, all the lexical components of the TTL language are represented be means of graphical templates. Due to the

software programming language constraint, as explained above, the graphical temporal operator templates are assigned a text string that corresponds to the name of the temporal operator. Objects of this class have common characteristics and they might differ in some others.

An example of a common characteristic that is considered when designing a temporal operator class object is that it must allow to be only connected to an object of class predicate. An instance variable is used to store the identity of the icons. For instance, the identifier of a temporal operator class object is "*TEMPORAL*". When trying to connect to another graphical template, a method implemented within the temporal operator class will ask whether the other object's identity is "*PREDICATE*"; if yes, it draws a line typifying the relationship between these two icons; if not, it rejects the connection and prompts an error message.

It has been clear that there is a close relationship between temporal operator icons and predicate icons. But the time bounds have not been yet defined. As said earlier, the predicate behavior in time is defined by the temporal operator and the time bounds. Not to increase the complexity of the iconic diagram of a TTLM, the time bounds are assigned to the predicate when the user is prompt to save it.

From the GUI design point of view all the temporal operators are exactly the same, except by the text string that corresponds to each temporal operator. But, from the TTL written language point of view, they differ in the implementation. In the language, the temporal operators have different effects over the predicates on which they are applied. For instance, the temporal operator HENCEFORTH makes the predicate to be true from the lower time bound onward. Another example is the temporal operator NEXT which sets the predicate to be true at the next time unit or at the next tick of the clock.

In Chapter 4 we will expand the graphical design of the temporal operators when the design of the GPE is described.

3.5 Temporal Logic Language

Icons, pictures, and symbolic graphics are sometimes considered as words or blocks which are placed in a two or three dimensional space following some predefined construction rules to create expressions and procedures in an iconic programming language. One of the goals of the GPE is to present to the user a set of graphical symbols that match the user's conceptual model of the constructs for the temporal logic language.

As we mentioned in Section 3.3 the temporal logic models are constructed from predicates, functions symbols, individual constants and variables, the logical operators, and the temporal operators. The language consists itself in a set of language components that we refer to as lexical components. The lexical components defined are: predicates, arguments, temporal operators, logical operators, and conclusion.

Each lexical component of the TTL language has its graphical representation. These graphical representations must be created and interconnected to build the iconic diagram that represents a sentence written in the temporal logic language. In interconnecting the graphical lexical components of the language some rules must be followed to preserve the correct syntax and semantics of the language such that the iconic diagrams are translated into a textual format that when parsed they meet all the parameters set by the parser which checks the syntax and semantics of the language as it was defined in Chapter 2. These rules are explained in the next section.

As in any other language, there must exist a mechanism to check for the grammar correctness. A parser is included as a way to check for the syntax and semantics of the language. The temporal logic sentences written in the graphical mode once translated into the textual format of our temporal logic language are parsed. A parser, which is itself an LR parser, checks the lexical components following some precedence rules that will define whether a reached lexical component is permissible or not. Those expressions correctly parsed will be understood by the reasoning mechanism.

3.6 Time Metrics in Temporal Logic

Temporal logic has suffered from its orientation toward eventuality rather than immediacy in real-time; indeed, pure temporal logic makes no reference to time. A temporal logic specification defines only invariants, eventualities, and order constraints on the sequence of states resulting from the execution of a DES without reference to when the states actually occur. But, the specification of DES typically depends on the specification of real-time properties.

The usual approach to specification and verification of systems is to enumerate all the transitions (events and actions), and the order in which the transitions can occur. For simplicity, quantitative timing constraints are left out so that the verification of system properties will not depend on when the events have happened. The challenge in real-time DESs is to re-incorporate a time metric.

Modelling time-dependent variables as explicit functions of time is very powerful for representing dynamic behaviors.

The concept of time metrics or counter is introduced as a means of keeping track on the system's states in time units. In Chapter 2, when defining the timed-temporal logic, a time variable is introduced. Although there might be more than one time variable, one for each system or part of the system, there is a common time variable that is used to maintain synchronized actions or event synchronization, that is, simultaneous participation of component events. The counter behaves like a clock, that is, it distinguishes time instants from each other.

Events have lower and upper time bounds. The counter prevents an event to occur before the counter has reached the lower bound time units and forces the event to occur between the lower and the upper bound. Computing time bounds is not a trivial exercise, usually a detailed knowledge of the hardware architecture is needed (as well as factors such as pipelining, memory and bus contention, and caching).

In [34], it has been suggested that real-time can be modelled in temporal logic simply as another global variable: the clock, and then assertions involving real-time will simply be temporal logic formulas involving the clock variable. The main problem with this approach is when to increment the clock variable in relation to the other activities in the systems. In Chapter 2 when the TTL language is described, a rule to increment the clock is introduced, this rule is as follows:

$$\Box[\neg EQ(\kappa, 0) \Rightarrow (\kappa = ADD1(\bigcirc\kappa))],$$

where ADD1(x) stands for that x is increased by 1.

In [31, 59], an automated tool is discussed for extracting timing information from real-time software. As an assembly instruction is generated by the compiler, the tool computes the execution time of that instruction as a function of the opcode, operands, and addressing modes, and the execution time is added to the current segment being built.

The simplest modelling tool for the states of the counter would appear to be the theory of finite state machines. To model the ticking of a clock, all that is needed at the highest level is single state label, say, *clockidle* with a single transition label *tick* that is a self loop from *clockidle* back to itself. If more information is needed, such as the current time in clock ticks, then *clockidle* must be broken down into many more substates each of which is needed to denote the completion of a different number of clock ticks (See Figure 3.2).

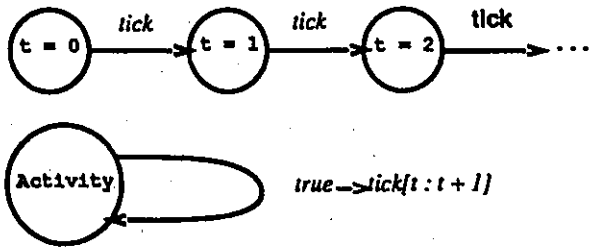

Figure 3.2: FSM of the clock.

3.7 Symbols of the Visual Programming Language

To write the temporal logic model using graphical symbols, five classes to create five different icons were needed. This class objects are:

- Argument (A),
- Predicate (P),
- Temporal Operator (T),
- Logical Operator (L), and
- Conclusion (C),

Arguments (A)

A predicate has a number of input parameters (arguments) over which it acts. Argument objects are created and assigned a name. Their representation is a plain rectangular box with a text string in the center. Since argument names vary from one predicate to another, a local menu defined in the argument class should provide an option to allow the user to change the name. An argument object can be connected to objects of classes predicate, and conclusion.

Predicates (P)

A predicate acts on a number of arguments, a temporal operator sets the timing behavior of the predicate, and that iconic structure is taken as an expression on which a logical operator is applied. Then, to represent those input and output parameters, some arrow class objects are attached. One downward arrow to represent the input for a temporal operator, an number of inward arrows that matches the number of arguments. This number is entered when the predicate is being created. And the outward arrow is used to connect

the resultant expression to a logical operator object. It has in the middle of the rectangular box a text string that represents the predicate name which is given when created.

Temporal Operators (T)

The design of the temporal operator objects was approached in the Section 3.4 when the design of the temporal operators was discussed.

Logical Operators (L)

The nature of a logical operator sets the minimum or maximum number of expressions a logical operator can handle. For instance, the logical operators AND and OR must be applied to at least two expressions; and NOT applied to only one expression. Hence, it was assigned only one inward arrow to take as many input expressions as permissible. The inference engine (described later in Chapter 5) is capable of reasoning over TTLMs with any number of expressions greater than one. Because they are chosen from a menu of logical operators, the name is assigned when being created.

An outward arrow is also attached. Through this outward arrow, a logical operator can be connected as an input expression to another logical operator (an iconic representation of nested logical operators). If a logical operator icon is the root of a TTLM, the outward arrow provides the connection to the conclusion.

A local menu of a logical operator class object provides an option to save a TTLM into a database. When this option is chosen, it must be firstly checked that this object is indeed the root. This is checked by asking if the identity of the object attached through the outward arrow is "$CONCLUSION$"; if not, it rejects the operation; if yes, the saving procedure starts.

When the saving procedure is being executed, it checks every logical operator AND or OR that forms the TTLM to see if the number of expressions attached to it is greater than

or equal to two. Any violation results in rejecting the saving operation and prompting an error message.

Conclusion (C)

A conclusion object is needed to express graphically the result if the premise holds true. Its design is very similar to that of a predicate icon in the sense that a variable number of arguments are attached to it and there is an inward arrow for each argument. A downward arrow is used to allow the connection with the root of the TTLM.

The characters between the parentheses are the textual notation used for the icons. The icons described above can be interconnected through two types of links:

- Data Path Connection (——), and
- Control Path Connection (▬▬).

The Visual Programming Environment can be compared with a formal language. A formal language comprises those particular sequences of symbols which satisfy certain grammatical constraints which thus define the language. It is defined in terms of an alphabet and grammar.

The alphabet is a finite set of symbols, and the grammar is a set of rules, or production rules, showing how the symbols in the language must be built up from the alphabet symbols in order to create the timed temporal logic models. The set of symbols, as described above, consists of {**A**, **P**, **T**, **L**, **C**, ——, and ▬▬}.

A temporal logic model consists of a number of logical operators (AND, OR, and NOR) whose statements are predicate names whose behaviors in time are described by a temporal operator. These predicate names have arguments who execute the action implied in the predicate names. There is also the need for a conclusion which has arguments as well.

The starting connection arrows of both data path and control path connections are

represented by arrows pointing outward on the graphical symbols. On the other hand, the ending connection arrows of the two types of connections are represented by arrows pointing inward and arrows pointing downward.

When a connection arrow in a graphical icon is the starting point of a connection, the connection is referred to the icon as an output control or output data path connection. When it is referred to the icon that sets the ending point, the connection is called input control or input data path connection.

As mentioned before, the grammar is a set of rules that together are used to generate the temporal language sentences from the graphical symbols of the alphabet. A formal definition of the grammar is shown below in Figure 3.3.

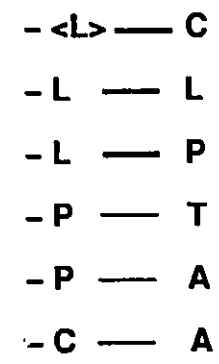

Figure 3.3: Formal Definition of the Grammar.

The graphical symbol in between angle brackets (L) means that this symbol has been designated to be the root of the temporal logic sentence. Hence, an object C is attached to it.

When in the formal definition of the grammar two graphical objects are related to each other through the double line (═), we mean that the protocol between these two graphical icons is by means of a control path connection, an example is the first item in the grammar definition. When the relation between two objects is established by a single line (—), the protocol is by means of a data path connection, for instance the last item of the grammar definition.

The grammar (a set of rules) is a scheme for generating sentences from the set of elements of the alphabet based on the formal definition of the grammar in Figure 3.3. Following are some rules that can help to understand better the meaning of the grammar:

- The root or the start symbol of a temporal logic sentence is in all cases a logical operator object **L**.
- A symbol **L** allows connections through its starting connection arrow to two types of graphical objects: **L** and **C**. It only allows for one output control path connection.
- The output control path connection of an object **L** which happens to be the root or start symbol must be connected to an object **C**.
- An object **L**, other than the root, must be output control connected to another graphical symbol **L**.
- The input control connection of an object **L** depending on the type of logical operator allows for as many connections as the user likes to graphical symbols of types **L** and **P**. For instance, if the logical operator is either AND or OR, it must take more than one, but if the operator is NOT it must just take one object either an object **L** or an object **P**.
- A graphical object **P** is connected to an object **L** through its output control connection arrow. Only one connection of this kind is allowed.
- A graphical object **P** is connected to an object **T** through its input data connection arrow that is pointing down. It allows only one connection of this kind.
- A graphical object **P** allows for as many connections as input data path connections of arrows pointing inward the icon has. The input data connections of an object of this type are done to objects of type **A**.
- Similar to an object **P**, a graphical symbol **C** is connected to as many objects of type **A** as necessary through the input data path connection arrows.

Figure 3.4 below shows a model of a rule built using the components of the language.

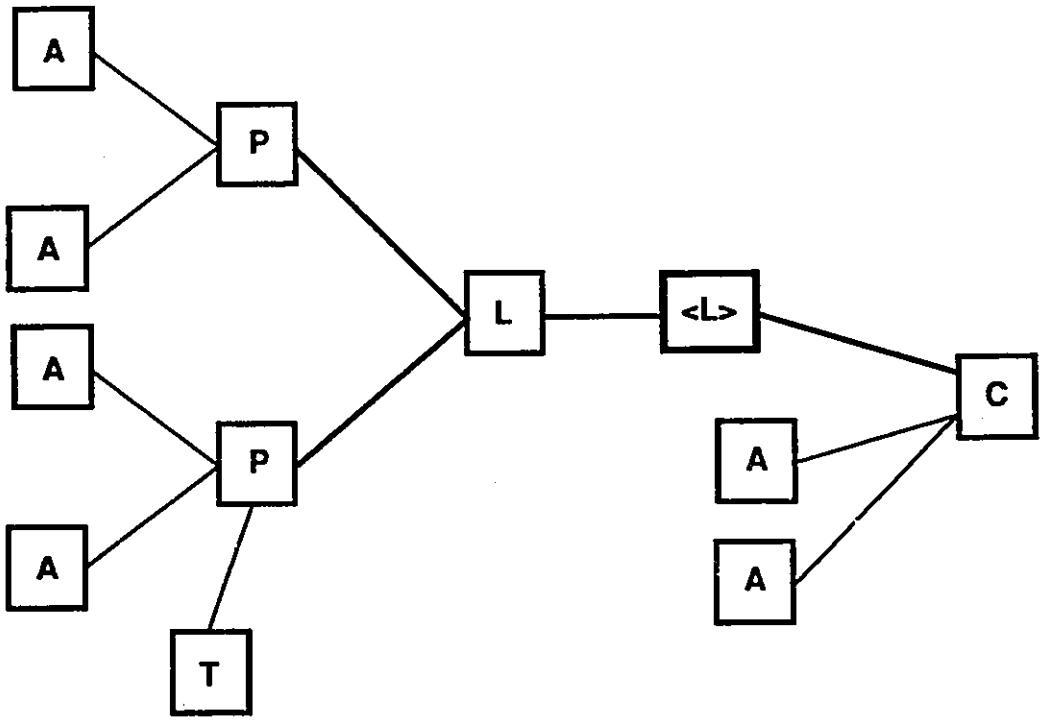

Figure 3.4: Icon Interconnection Using the Language Components.

The iconic representations of the lexical components of the timed temporal logic language are presented in the next chapter in which their implementation is described.

3.8 Conclusions

In this chapter we introduced some issues on GUI design for a timed temporal logic language .

We have briefly described the complex architecture of the GPE for simulation of real-time DESs. It was divided into three blocks with their own classes interacting with one another. The class LogicalCls serves as the interface in the GPE architecture, and furthermore, it creates and initializes the working memory from database files, and creates the menu bar from which the features of the GPE are chosen.

We have also described the design of four temporal operators: NEXT ($\bigcirc$), HENCEFORTH ($\Box$), EVENTUALLY ($\Diamond$), and UNTIL ($\mathcal{U}$); that together with the global clock timed temporal logic models for DESs can be constructed.

A global clock was implemented as a way of overcoming the lack of temporal operators to make reference to time since the specification of DESs typically depends on the specification of real-time properties.

The design of each of the graphical symbols for the TTL language was introduced and some essential menu options were described.

Finally, an introduction to the visual programming language was given. The visual programming language will be extended in the next chapter when the design of the GPE is approached.

Chapter 4

Designing the GPE, The Timed Temporal Logic Language and the DES Reasoning Tools

It is commonly accepted that graphics present to the user a visual syntax that matches the user's conceptual model of the constructs of the language. In this chapter we introduce the design methodology to build the visual programming environment. The features of the GPE for doing simulation of DESs are also described.

4.1 Introduction

4.1.1 Object-Oriented Paradigm

The object-oriented paradigm has five key components: object, message, class, instance and method. Programming in this paradigm involves creating a set of objects with the proper methods that will be invoked at the appropriate time through message passing among these objects. An object-oriented language comes with its own abstract classes of objects which together form a programming environment.

Two of the OO Programming properties are encapsulation and inheritance. Inheritance provides a flexible programming environment that is organized in a hierarchical structure of object classes with reusable programs. Encapsulation is used to combine both the data and the operators into a unit called an object.

4.1.2 GUI Design

Applications developed for standard graphical user interfaces are becoming more and more popular. Object-oriented programming reduces the complexity of designing a visual programming environment by encapsulating standard windowing behavior and visual language components behavior into predefined class objects.

In Chapter 3 some issues on GUI design were introduced. A GUI should comply with three requirements:

- Completeness. A GUI must provide the on-line creation of all the objects needed to write applications, it must allow the user to run the applications by mouse-driven events, and it must provide supporting or maintenance tasks.
- Convenience. It must be user-friendly, self-explanatory, and error-resistant.
- Compatibility. The design and underlying data structure should be designed so that it could be easily future-extended to handle new objects and information.

It will be now shown how these design requirements were met by the GPE implementation.

According to the design methodologies described above, the GPE is provided with a layer hierarchy which provides the system being developed with a series of layers that interact with one another to build up the interface. One of these layer is the menu layer which is attached to another layer that acts as the backbone layer of the GUI.

The menu layer must be designed is such a way that it allows the creation of all

the necessary objects for the application. Not only do objects have to be created, but also it should support the environment with some menu options from which the system can run some tasks as well as some maintenance routines, such as clear the screen at any time. For the particular case of this thesis, the menu is a bar menu from which the graphical representation of the lexical components of the visual language can be created and displayed on the screen by means of mouse-driven events. There is an option in the bar menu to create each lexical component. Since information is being stored in several databases, this bar menu layer allows the user to retrieve the content of those databases at any time. Since the GPE is aimed at running simulations of DESs whose models are written using a TTL language, there is an option to run the simulation process. This option pulls down a submenu from which the user can display on the screen the results of the simulation previously executed. And, it also allows the user to perform some maintenance tasks such as clear the screen, save an iconic structure in a file, and retrieve such a file.

Since the iconic representation of TTLMs can be complex and takes much of the working layer, a horizontal and a vertical scroll bar have been added allowing for more space on which the TTLMs are built.

Graphical environments have an inherent complexity that makes them good candidates for object-oriented programming. Many elements can be conceived as objects that are made up of both data and functionality. For instance, a generic object includes private data to maintain its location, size and built-in functionality so that it can be moved, resized, etc. More formally, this process is known as inheritance allowing the user subclasses or descendants of existing kinds of objects to acquire their parents' methods and instance variables through the so called class hierarchy.

In the Objective-C environment most of all layers inherit their characteristics from ancestor classes. For instance, the class Menu from which the bar menu is created inherits all the layer capabilities from the classes Layer, BorderLayer, and LayerMedium, the capability of having a text string in each of the menu option is inherited from the DispMedium and DispObject classes. The StdLayer class also inherits variables and methods from the classes

Layer, BorderLayer, LayerMedium, DispMedium, and DispObject. It is obvious that the graphical capabilities of the different layers are the same inherited through the inheritance hierarchy that the object-oriented paradigm provides.

The design of the GPE has been focused to be amiable to the user. For a first time user, it might seem difficult to use, but the icons of the lexical components of the language are designed in such a way that they visually have a meaning so that they can be interconnected in a self-explanatory fashion to form the TTLMs. For instance, a predicate icon has a number of input arrows that imply that some input parameters are needed. In addition, the GPE has been provided with a number of messages to tell the user when a connection is wrong, this messages also tell the user why the connection is not allowed.

The underlying code of the GPE has been designed in a modular fashion, such that, when new features or enhancements are added, it is necessary to add only one line of code to the main GPE program. This line sends a message to the method which implements the new feature or enhancement to the recipient.

In the sequel of this chapter, we will describe the classes implemented to accomplish the design methodology that was described above.

4.2 The Visual Environment

ICpak201, one of the libraries of Objective-C, provides layer hierarchy management, displaying, and event handling through which visual environments are built. The Objective-C layer hierarchy can be run in different environment such as sunviews, openlook, and xwindows.

4.2.1 Display Objects and Display Mediums

A Display Object is an object that may be displayed onto some other objects. These objects where Display Objects can be displayed on are Display Medium objects. Examples

of Display Objects are a line, text, or pixel image. Examples of Display Mediums are screens, bitmaps, and layers. Figure 4.1 shows an example of a Display Object on a Display Medium.

The Display Medium class differs from a Display Object class in that an instance of the Display Object class may be displayed onto something but cannot have anything displayed on it.

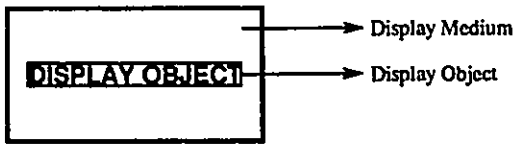

Figure 4.1: Display Objects and Display Mediums.

4.2.2 Layers

Layers are stacked on top of one another to construct the layer hierarchy. It is this layer hierarchy that defines how the visual environment looks and reacts to user input, as shown in Figure 4.2.

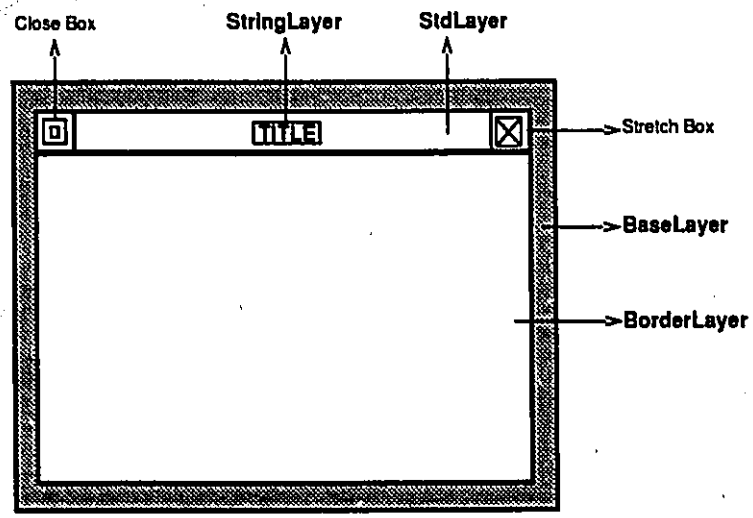

Figure 4.2: Layer Hierarchy.

There are a few different layer classes. Among those layer classes are BorderLayer,

StdLayer, StringLayer, and BaseLayer. BorderLayer is a layer with a border and an opaque inside color. StdLayer is intended to be a base from which applications can be built. A StdLayer provides a close box, a stretch box, and a title. A StringLayer is used when a text string needs to be attached to a layer or when the text string has to be installed into the layer hierarchy. The root of the layer hierarchy is an instance object of the BaseLayer class, which provides both an interface for an instance of the class Display and the controller behavior of the layer hierarchy.

4.2.3 Menus

Menus are created by first building a logical menu. The logical menu is then formatted into physical menu. The Menu class is used to build logical menus. Each instance of Menu is a node in the logical menu tree structure.

The logical menus specify the items in the menu, as well as the submenus. The logical menus are arranged in a tree structure, where the children of a node are the items in the node's menu. If the child node has children, then that child node has a submenu with its children as the items in the submenu.

The BarMenu class is a physical menu formatted horizontally with no title, visible and usable only when its back layer is visible. Another kind of physical menu is the PopUpMenu class, which appears in response to a user action, typically by the pressing of a mouse button. These two menu types are shown in Figure 4.3

The AdHocMenu class is used to build physical menus. The submenu of an AdHocMenu instance may depend on some dynamic state of the software system, or on the entries in a database. The AdHocMenu does not decide which items are in the menu until just before it is called on to display itself.

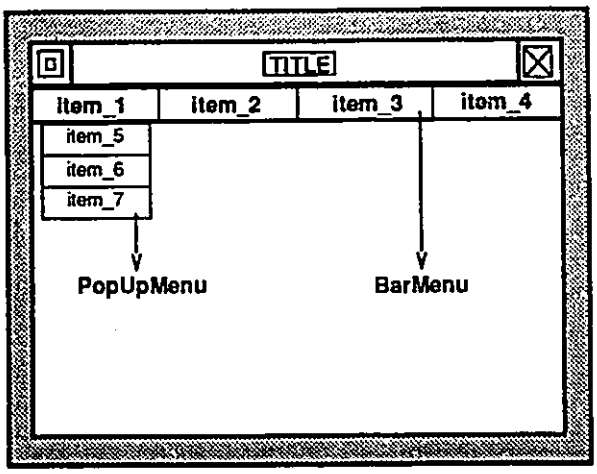

Figure 4.3: Types of Menus.

4.2.4 Events

The EventQ class manages the input event queues and queries to the state of the button and events. Various kinds of events are generated in the execution of an application. Pressing or releasing a mouse button, pressing or releasing a key on the keyboard and moving the mouse are the user actions that generate events.

As events are generated, events are added to a first in first out queue. EventQ is the interface to this queue. It provides the ability to ask if there are any events in the queue, to obtain the next event from the queue, to put an event into the queue, and to flush the queue of all events. It also provides access to the last event from the queue.

4.3 Graphical Symbols for Programming of Simulation Models

Visual Interactive Simulation (VIS) is a term that has been used in connection with a simulation program which has features for graphical creation of simulation models (graphical programming), dynamic display of the simulated system (visual output), and the user interaction with the running program (user interaction).

4.3.1 Graphical Programming Symbols for the TTL Language

The design of the graphical symbols was discussed in Chapter 3. Using the graphical programming facility, TTLMs for simulation of DESs are created on the screen in an interactive style by interconnecting the graphical symbols. A set of graphical symbols, according to the semantics, are necessary: Arguments, Predicates, Temporal operators, Logical operators, and Conclusion.

Each element of the above set of objects is implemented by a different class. The icons are visual representations of their respective classes. Therefore, by properly interconnecting the instances of these classes, and by manipulating them in the work-space, the behavior of the system being simulated is built using the TTL language.

Each icon corresponding to a component in the TTLM is represented in the visual environment as a rectangular box containing a name. The graphical representation of each component of the TTLMs can vary slightly from one another. Icons belonging to different classes have a number of common features, such as they can be moved anywhere within the work-space, they display a submenu by pressing the right mouse button, and they are connected to other icons in a meaningful way.

4.3.1.1 Arguments

The iconic representation of an argument is a rectangular box with a text string in the center corresponding to the argument name. Initially the text string is "Arguments" which can be easily changed to another text string given textually by using one of the menu option supported by the popup menu display when the right mouse button is depressed while the cursor is on the icon. The popup menu is shown in Figure 4.4.

A menu that allows the object to perform local operations is created. This local menu allows the user to quit or delete an icon from the screen as well as to remove the connection through which the argument is attached to an icon of class PredicateCls. Since argument

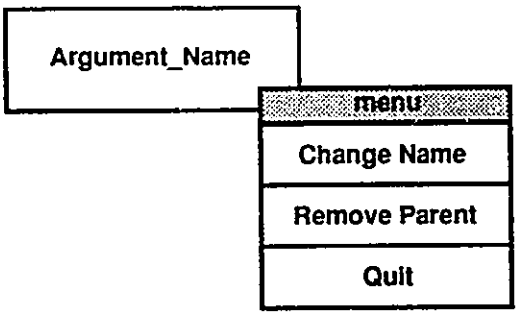

Figure 4.4: Graphical representation of Arguments class objects.

names vary from predicates to predicates, a handy option to change the name is included.

An argument is connected to a PredicateCls class object. No Arrow class object is needed since this is the only possible connection an argument object allows. An argument name is connected to a predicate object through data path connections explained in the next Section.

4.3.1.2 Predicates

The graphical representation of these objects is a rectangular box and a predicate name attached in the middle of it. They have several Arrow class objects attached as well as a means of inferring the need of input and output, see Figure 4.5.

Following very closely the design steps described earlier in this chapter, some arrow icons are being used to indicate that there are input parameters and a result. Three different types of Arrow class objects are attached to a Predicate object, they are: arrows pointing inward, downward, and outward from the predicate name. The number of arrows pointing inward attached depends on the number of arguments a predicate name can take. For example, if a predicate has three arguments, when created, it will display three arrow objects pointing inward the predicate name.

The other type of arrow object attached is the arrow pointing downward and it is used to represent or allow the connection of a temporal object to a predicate. And the last one

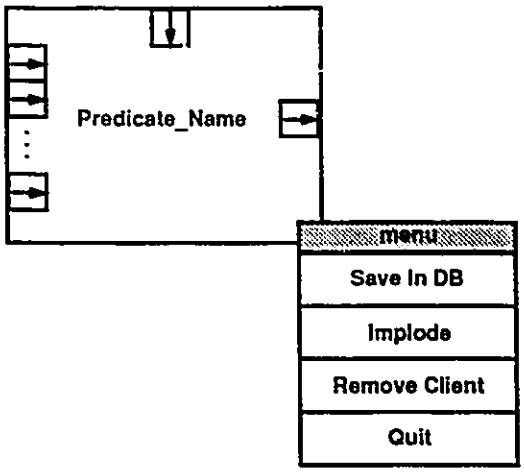

Figure 4.5: Graphical representation of PredicateCls class objects.

is the arrow pointing outward to typify that a result is coming out and has to be directed to another icon.

A selection menu is also shown in Figure 4.5. It allows the user to save a statement in a database for future modifications or reuse, to reduce the whole iconic diagram corresponding to the statement into a single box, to remove a connection to any of the icons attached to the predicate, and to remove the predicate's graphical representation from the screen provided that it is not connected to any icon at all.

4.3.1.3 Temporal Operators

A temporal operator, in the language defined in Chapter 2, is a lexical component attached to a predicate. This temporal operator sets the behavior of the predicate in time along with the time bounds.

A temporal operator's graphical representation is very similar to that of an argument's, except that when created, the temporal operator's name is already assigned to the rectangular box since they are chosen from a menu whose content is all the system's temporal operators.

To make them somehow visually representative to the user's eyes, they are created with

the text strings of the operators they represent in the middle of the rectangular box. For instance, consider the temporal operator NEXT, shown in Figure 4.6, that is represented by the rectangular box and the text string NEXT in the center.

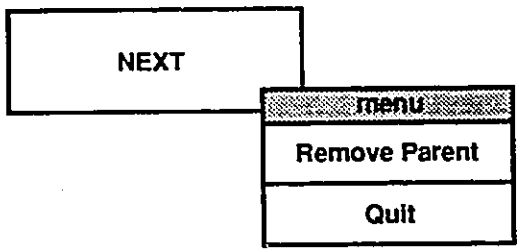

Figure 4.6: Graphical representation of Temporal Operator class objects.

Temporal operators are connected through data path connections to objects of PredicateCls class.

4.3.1.4 Logical Operators

Its graphical representation is a rectangular box with the name of the logical operator in the center and two Arrow class objects attached. The logical operator name is assigned when created and as of temporal operators, they are chosen from a selection menu. The arrow pointing inward or input takes the statements that will eventually lead to a truth value. The graphical representation of a logical object can be seen in Figure 4.7.

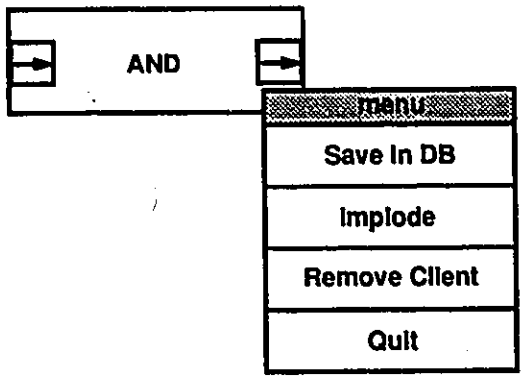

Figure 4.7: Graphical representation of Logical Operator class objects.

The inward and outward arrows are called input connection and output connection, respectively. An input connection is through which the input parameters are connected to the LogicalCls object. If the logicalCls object is "NOT", the number of input parameters to be connected is one; and if it is either "AND" or "OR", there is no bound for the number of input parameters, but must be greater than one. An input connection can be a connection from either an object of class PredicateCls or an object of class LogicalCls when the premise if composed of nested logical operators. At the same time the outward arrow serves as an interface interconnection with icons of class LogicalCls (when premise involves nested operators) and with icons of class Conclusion when the logical operator is the root of the premise, that is, the conclusion's attaching means the termination of the rule's construction. A LogicalCls class icon is the root when it is connected to a Conclusion class icon.

The popup menu corresponding to icons of this class allows the user to save the complete graphical structure of the rule provided that the root has been chosen and the conclusion has been attached to the root. The user is allowed to close the graphical structure into a small box when iconic diagrams are getting too big and the working space becomes critical. And the other two options are to remove any connection as well as to remove the icon from the screen.

4.3.1.5 Conclusions

A Conclusion class icon is very similar in both the creation method and the physical appearance of the icon to that of the PredicateCls class icons. Figure 4.8 shows a Conclusion class icon.

When created, a conclusion icon is given textually a string that represents the conclusion name. It also has a number of inward arrows that is set by the number of arguments that are connected to that conclusion name. On top of the inward arrows column there is a downward arrow that is used to symbolized the connection with logical operator that

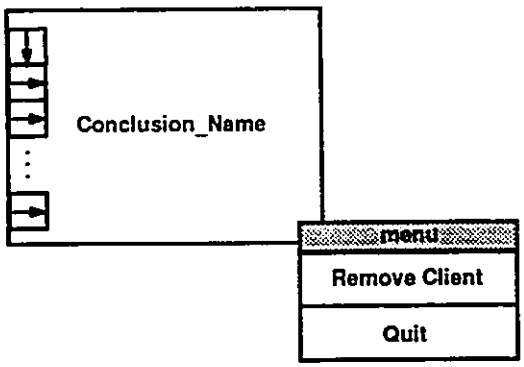

Figure 4.8: Graphical representation of Conclusion class objects.

represents the root. The popup menu allows for two basic operations: remove a connection and remove the icon from the screen provided all the connections have been removed.

4.3.2 Graphical Representation of Connections

Two types of connections have been defined in the visual programming environment: Data connections, and Control connections.

Connections are used to provide the programmer with information about the input and output parameters as well as information about linkage to some other icons or data path. A data path connection is referred to an icon as input data path connection or input control path connection if the connection ends on any of the arrow objects attached to the icon. Similarly, a connection referred to an icon is an output data path connection or output control path connection if the connection starts from the icon and ends in any of the arrow objects of another icon.

In a graphical environment, many ways to distinguish between data and control connections can be used. We have chosen to represent data connections as a single line and control connections as double lines. To draw a line, the left mouse button is depressed and with the button depressed, the mouse device is dragged to the destination icon. When the mouse device in over the destination icon, the line will be drawn.

4.3.2.1 Data Connections

The data connections are used to link input parameters with some other icons following some connection rules. An example is shown in Figure 4.9 where the English expression *receiver has received packet number zero* is represented. RECEIVER and PACKET0 represent the input data to the predicate icon RECEIVED.

In order for the user to know how many input data parameters any icon has, a number of input arrow objects equal to the number of input data parameters of the predicate are attached to it. This number of input parameters a predicate can have is defined when the predicate is created to be used or stored in the database. An input arrow object is a dummy icon attached to another icon only for visual representation, hence, it is transparent to the mouse generated events.

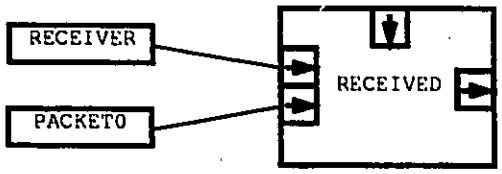

Figure 4.9: Data connections of input parameters.

Data connections are destroyed by means of user actions in two different ways; by selecting either the submenu option *quit* to delete the icon, or the submenu option *remove* to remove a connection to a user-selected icon.

4.3.2.2 Control Connections

As mentioned in Chapter 3, TTLMs are composed of nested logical operators, each of them taking one, two or more statements depending on the logical operator. A statement is the temporal logic model component formed by a predicate, its arguments, and a temporal operator. For instance, consider the expression:

AND(*statement 1*)(OR(*statement 2*)(*statement 3*))

that represents the premise of a temporal logic model or rule.

To establish that sequence, a special kind of connection different than the data connection must be defined. Control connections are used in the acyclic graph that represents the temporal logic model to define the flow chain or path which each icon follows when the iconic diagram is transformed into textual form (See Figure 4.10).

The input and output arrows of control connections are attached only to objects of the class LogicalCls which creates the graphical representation of the logical operators.

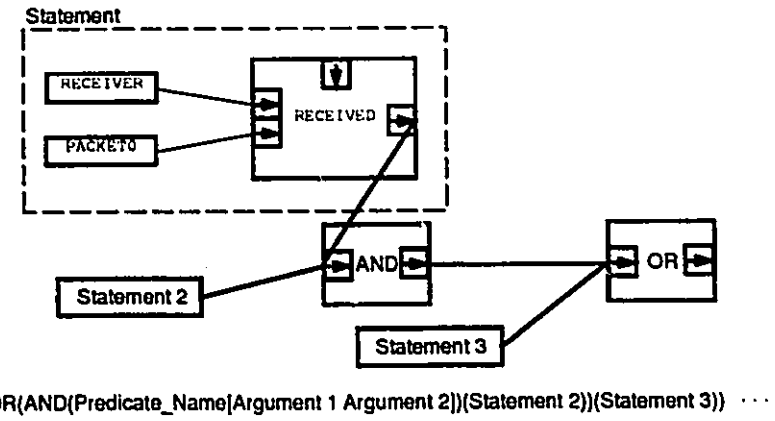

Figure 4.10: Control connections for path definition.

As part of the support classes a class called **Implode** is created to allow the user to encapsulate a complete temporal logic model into one single box when an iconic diagram is getting too complicated to be handled on the screen.

4.3.3 Dynamic Display of Simulated Systems

The dynamic display facility portrays the dynamic behavior of the system components on the screen as animated icons. This capability provides the user with an insight into the current state of a simulation model. Based on that, the user is able to observe certain problems in the model and make necessary changes by interacting with the running system. Interaction can be such that the simulation halts and requests information from the user,

or the user stops the simulation at will and interacts with the running program.

To achieve animation for displaying the simulation models, three classes have been created, they are the StateCls, States, and Events classes. The class StateCls is the only one that is visible to the user which uses the Events and States classes as support to model the system behavior.

Each instance of the StateCls class represents a state which the system is at certain point in time during execution. They are graphically represented as a rectangular box with a centered text string as the process name. No instance of this class can be user-created. They are created according to the specifications of the system behavior given by the temporal logic knowledge, rules and facts, that after created they were stored in a database.

Since states are not isolated one another, a connection is established. A system can move from one state to as many as its behavior establishes. A connection path is used to represent the different ways the system can behave, not to represent any data connection or control connection whatsoever, providing the user with a sort of graphical sense of guidance. Figure /reffigure414 shows the state transitions of the example described in Section 2.5 in which the specifications of the Alternating Bit Protocol (ABP) were written using TTL described in Chapter 2.

An instance of the State class has an instance variable called "events" that is an ordered collection of Events class instances. In this ordered collection, all the events that can possibly occur at that state are stored. Events are instantaneously occurrences that trigger state transitions.

As said before, the Events and States classes are supporting classes and are not displayed on the screen. The Events, States, and StateCls classes work together very close. An instance of the Events class has an instance variable of the class States which stores the next process reached if that event occurs.

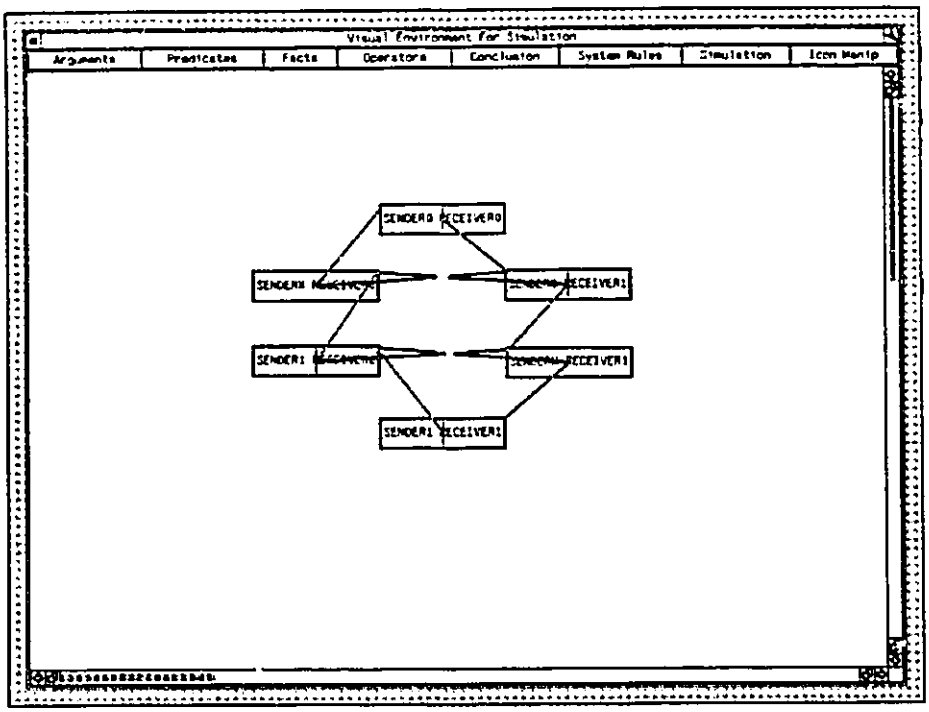

Figure 4.11: State transitions of the ABP.

4.4 DES Reasoning

To deduce conclusions from the graphical TTLMs, the TTLMs expressed by predicates combined with logic connectives and temporal operators are evaluated and a truth value is generated based on the given facts.

4.4.1 Parsing

The TTLMs are first written in the textual format of the language. A parser is required to check the syntax and semantic of the language as it was defined in Chapter 2.

A parser is needed to check for correctness in the syntax of the textual format of the TTLMs. The parser analyses the TTLMs in the following way: first combining word 1 and word 2, then adding word 3, then adding word 4, etc., until there are no more words. The parser is based on the notion of possible continuation; after word n has been added, the rules imposed by the grammar specify what the categories of word $n + 1$ might be.

The parser used in this work is described in [57]. It was built by a research team at Iowa University led by Dr. Teodor Rus.

4.4.2 Evaluating

An inference engine has been built to take on the evaluation of the TTL rules that specify the system (See next chapter). The rules are evaluated and a truth value issued according to a group of facts. In the evaluation process two mechanisms of inference are used; these inference mechanisms are: forward chaining and backward chaining (See Algorithm 4.1). The behavior of the DESs depends mainly on the chosen reasoning mechanism that leads to the conclusions. Choosing the inference mechanism can be critical in rule-based systems since some aspects are best handled by an inference strategy and some other aspects are best handled by a different inference strategy.

The mechanisms of inference implemented in this thesis are forward chaining and backward chaining. The algorithms for these two methods are shown in Algorithms 4.2 and 4.3, respectively.

Forward chaining is the strategy of searching from the present state toward the conclusion. A goal is taken and the database of facts is explored forward down until a fact that matches the goal is found.

In the backward chaining strategy, the search is performed from the conclusion back to the initial node or premise. When backward chaining method is performed, a goal is matched against the conclusions of the rules store in the database of rules. When the goal matches a conclusion, this rule is taken for further reasoning.

Algorithm 4.1: Implementation of rules evaluation.

1.- BEGIN. Take premise of the rule.

2.- Read an element of the premise.

2.1.- IF element is "(" THEN increment the counter.

2.2.- IF element is ")" THEN decrement the counter.

2.2.1.- IF counter is 0 THEN go to step 4. OTHERWISE, go to step 2.

2.3.- IF element is a statement THEN send the statement to forward chaining (See Algorithm 4.2).

2.3.1.- IF return value is true THEN go to step 2. OTHERWISE, send the statement to backward chaining with the statement as the goal (See Algorithm 4.3).

2.3.1.1.- Go to step 2.

2.4.- IF element is a Logical Operator (i.e. AND, OR, or NOT) THEN go to step 3.

Step 3 through step 7 correspond to the method called AND, OR, or NOT depending on the Logical Operator that was read from either step 2.4 or step 3.4.

3.- Read next element from the premise.

3.1.- IF element is "(" THEN increment the local counter.

3.2.- IF element is ")" THEN decrement the local counter.

3.2.1.- IF the local counter is 0 THEN end evaluation and return truth value to calling method. OTHERWISE, go to step 3.

3.3.- IF element is a statement THEN send the statement to forward chaining.

3.3.1.- IF return value is true THEN go to step 3. OTHERWISE, send the statement to backward chaining with the statement as the goal.

3.3.1.1.- Go to step 3.

3.4.- IF element is a Logical Operator (i.e. AND, OR, or NOT) THEN go to step 3 recursively.

4.- END. Send truth value to calling method.

Algorithm 4.2: Implementation of forward chaining.

1.- BEGIN. Take goal passed from calling method.

2.- Set pointer to beginning of the database of facts, a counter to zero, and the truth value to false.

3.- Compare the goal to the fact the pointer is pointing to.

3.1.- IF goal matches the facts in the database THEN set the truth value to true and go to step 5. OTHERWISE go to step 4.

4.- Increment pointer.

4.1.- IF pointer is less than or equal to size of database of facts THEN go to step 3. OTHERWISE, go to step 5.

5.- END. Return truth value.

Algorithm 4.3: Implementation of backward chaining.

1.- BEGIN. Take goal passed from calling method.

2.- Set pointer to beginning of the database of rules, a counter to zero, and the truth value to false.

3.- Compare the goal to the conclusion of the rule the pointer is pointing to.

3.1.- IF goal matches the conclusion of the rule THEN take that rule and send it to the evaluation process (Algorithm 4.1). OTHERWISE go to step 4.

3.1.1.- IF returned value is true THEN set truth value to true and go to step 5. OTHERWISE, go to step 4.

4.- Increment pointer.

4.1.- IF pointer is less than or equal to size of database of rules THEN go to step 3. OTHERWISE, go to step 5.

5.- END. return truth value.

4.4.3 Reachability Analysis

The specifications of real-time DESs are given in TTLMs. When reachability analysis is performed, there is a need to track the computation as a trajectory progressing in time, analyzing the dynamic behavior of the TTLMs such as reachability properties. The dynamic behavior of DESs is characterized by transitions from one state to another state such transitions are triggered by enabled events. The enable condition of events depends upon the truth value of the TTL formulas describing the dynamics.

The dynamic of DESs is described in TTLMs. These TTLMs are inspected and the reachability set is constructed. Lin in [36] introduces an algorithm to compute the reachability graph for DESs whose behavior has been written using temporal logic models. The reachability graph consists of a set of all the reachable states and the enabled event set for each state.

The Algorithm mentioned above has been used here to calculate the reachability graph. As a Visual Programming Environment is been developed and the reachability graph needs to be displayed on the screen, a new class is created to graphically represent the states of the reachability set so that they can be displayed on the screen along with all the connections that represent the state transitions one state to another provided that such transitions have been triggered by permissible events. In Chapters 5 and 6 some examples are shown to illustrate the reachability graph of DESs.

```
(1):  -statesBuilder {
(2):    [self initialState: stateBase];
(3):    for(i = 0; i < [stateBase size]; i++) {
(4):      aState = [stateBase at: i];
(5):      for(j = 0; j < [[aState events] size]; j++) {
(6):        nextState = [States new];
(7):        [nextState initializeState: [[[aState events] at: j] nextState]];
(8):        if([nextState isInDataBase: stateBase] == NO) {
```

```
(9):            [nextState eventsExtractor];
(10):           [stateBase add: nextState];
(11):         }
(12):       }
(13):     }
(14):     return self;
(15):  }
```

The above method is implemented in the class "States" and is responsible for constructing the reachability graph. When called, It first sends the message "initialState" in code line (2) to the receiver. This method finds the initial state from the database of facts and stores it into the database of states "stateBase".

When the first "for" loop is entered, the size of "stateBase" is 1 and only contains the initial state which is the initial condition, from that state the reachability graph is built. In line (4), a variable of class "States" is assigned the element of "stateBase" the variable "i" is pointing to. Then, all the events that can be triggered from that state are found, and from those events, the corresponding next states are found in code line (7) and if the do not yet exist in the database "stateBase", the are added into it (code line (10)).

4.4.4 Monitoring

According to the specifications of the required behavior, there are undesirable states and events in the uncontrolled system. These states and events should be avoided in order to preserve the well being of the system.

A monitor, following an idea from [36], is implemented to give a list of warnings on which states should not be reached and a list of suggestions on which events should be disabled to guide designing a controller with the undesired behavior eliminated. What follows is the actual implementation of the method *checkBehavior* that does the monitoring to prevent the undesired behavior and summit suggestions accordingly.

```
(1):  -checkBehavior {
(2):    for(i = 0; i < [behaviorBase size]; i++) {
(3):      aState = [behaviorBase at: i];
(4):      anEvent = [[aState events] at: 0]; // list of events of aState
(5):      for(j = 0; j < [stateBase size]; j++) {
(6):        aName = [aState stateName];
(7):        if([[[stateBase at:j] stateName] isEqualSTR:[aName str]]==YES) {
(8):          returnInteger = [anEvent checkEventIn: [[stateBase at: j] events]];
(9):          if(returnInteger >= 0) {
(10):           [[[[stateBase at: j] events] at: returnInteger] occurrence: NO];
(11):           aNextState=[[[[stateBase at:j] events] at:returnInteger] nextState];
(12):           theNextState = [self returnState: aNextState];
(13):           if(theNextState != nil)
(14):             [theNextState reachable: NO];
(15):         }
(16):       }
(27):     }
(18):   }
(19):   return self;
(20): }
```

The above method basically shows how to determine which states are unreachable and which events are disallowed as well as from which states they should not occur. A database whose content is the undesired behavior is kept in the variable "behaviorBase".

In code line (3) an element of "behaviorBase" is stored in the variable "aState" and in code line (4) a list of events from "aState" that should be disallowed is stored in anEvent. The object "aState" is compared with the states in the database stateBase. When a matching one is found, the list of events "anEvent" is checked against the list of events from the matching state (code line (8)). Whatever matched events from these two lists are

categorized as disallowed by storing the boolean value "NO" in the "occurrence" variable. In line (12) the message "returnState" is sent to the receiver "self", if the return value is other than "nil" (code line (13)), the state is unreachable, otherwise, is reachable.

After the above method is run, the events at each state and the states are inspected and the list of warnings and suggestions are recorded into a file which can be edited from the VPE allowing the user to see the recommendations without having to quit the VPE.

4.4.5 Optimizing

The events in the event set are given a cost value. An optimal path is produced according to the corresponding sequence of events which drives the system along the optimal path from the initial state to the final state with the minimum of the cost [36].

The optimal path is achieved using the algorithm A^* ([60]). This algorithm uses the uniform-cost procedure finding the exact distance from the start node to the reached node. In addition, the A^* algorithm also considers some heuristic information about the distance from the current state to the goal state.

Every event in the event set is assigned a cost value. Every state is assigned a value that is the true distance between that state to some goal state. The A^* algorithm opens state nodes in an order that gives highest priority to nodes likely to be on the shortest path from the initial state to the final state. To do this it adds the cost of the best path found so far between the initial state and the current state to the cost of an event that triggers the transition from the current state to an intermediate state, provided that this cost does not exceed the true distance from the current state to the final state.

When the optimal path is found, it is immediately reflected on the screen. The trajectory that leads the process from the initial to the final state is shown in the reachability graph that is on the screen.

4.5 Conclusions

In this chapter we have discussed the programming environment used to build the visual interface. Object-oriented programming enables the visual environment programming to be written with a focus on the description of the problem.

Graphical icons when properly selected and applied, can provide an excellent basis for building TTLMs for simulation of DESs. They represent the user's abstract interpretation of real problems.

We have designed a dialog-based programming environment in which the TTLMs for DESs are graphically built.

The design methodology for the inference engine was introduced. A parser is used to check the grammar of the language, and two mechanisms of inference are implemented to reason about the behavior of the real-time DESs: forward and backward chaining.

A new class was introduced to graphically represent the states of the reachability set. The reachability set is produced when the TTLMs are inspected by the inference engine. A monitor is implemented to give suggestions to build a controller eliminating the undesired behavior of the system.

As part of the visual environment for simulations, we have introduced an animation feature to the simulation models. It is a valuable support for the user-model interaction since most malfunctions of the model are apparently visual. The user is able to accurately monitor the actions of the systems and intervene, when needed.

It has been suggested as a future enhancement that a class able to create icons that handle the control path connections be implemented. This icon will contain the text string "IF( )" in the middle and one inward and one outward arrow. The inward arrow is to take as an input parameter the premise of the TTL rule being built and the outward arrow to direct the rule construct to the conclusion.

Chapter 5

Implementation of the GPE and TTL Classes

A visual programming environment is based on a set of icons that are used for pictorially representing conceptual entities and operations. In this chapter we describe the classes implemented to build the graphical representation of the lexical components of the language as well as some examples on how to create TTLM. A description of the inference engine and the visual simulation is also given.

5.1 Introduction

A programming environment for a specific visual language is generated by specifying the syntax and the semantics of the language. In the environment generated, the programmer is provided with an integrated tool-suite to support a broad spectrum of the software life cycle.

In the object-oriented paradigm, objects can be defined into different kinds of classes. Objects created from each class will be similar but not necessarily identical. This is actually a higher level of abstraction and a more natural way of programming than it is possible with procedure-oriented simulation languages.

Object-oriented simulation programs make excellent use of modern software engineering concepts, such as modularization, extensibility, incremental, and exploratory style of programming. This style of programming will be one of the essential characteristics of a rapid model development environment for complex simulation problems.

5.2 Graphical User Interface

The LogicalWS generates a work space upon which temporal logic models are built. The user, interactively, creates the objects that represent the lexical components of the TTLMs. Once placed on the screen, the objects can be moved anywhere within the screen's bounds by dragging them with the mouse device.

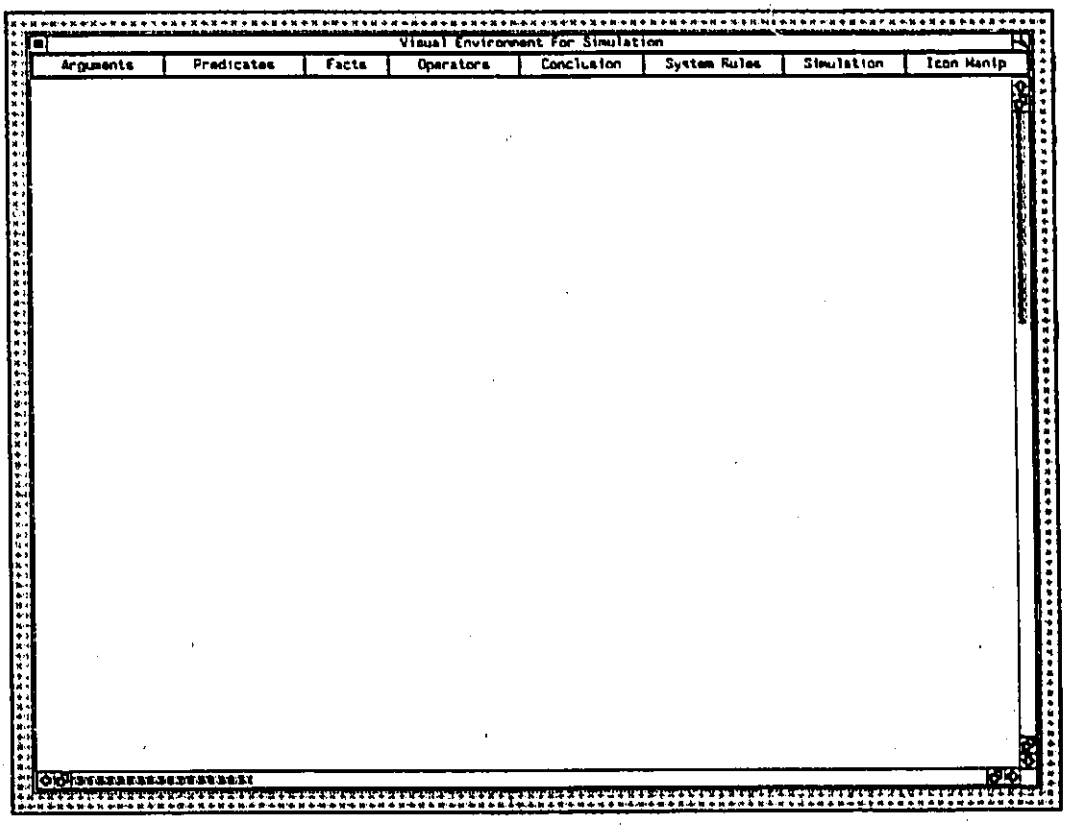

Figure 5.1: Graphical User Interface.

5.2.1 ArgumentsCls Class

The arguments are created through the class ArgumentsCls. They are dynamical structures that hold a text string, the argument name, that will eventually be connected to either an icon that belongs to the PredicateCls, or FactCls, or Conclusion.

To create an object of this class it suffices to click the left mouse button while the cursor is inside the first menu selection, *Arguments*. At first when created, an argument object has the text string "Arguments". By pressing the right mouse button on the object, the graphical environment will pop up a menu, among the options are: *Quit* and *Assign Name*. When the object is connected to any allowed object, the visual environment will display a slightly different menu with the following options: *Quit*, *Assign Name*, and *Remove Parent* connection.

To connect an icon of class ArgumentsCls, the left mouse button must be kept pressed while dragged until the cursor reaches the destination icon which can be of any kind among those mentioned above. Once the mouse button is released a line is drawn indicating a sort of relationship between the ArgumentsCls object and the one just connected to it. This icon can be moved all over the screen maintaining at any time its connection lines to other icons.

5.2.2 PredicateCls Class

These class objects are created using the second menu selection from the bar menu. This option assists the user in creating PredicateCls class objects. The submenu options are shown in Figure 5.2. Objects of both classes, ArgumentsCls and TemporalCls classes, are the only ones allowed to be attached to an icon of class PredicateCls by means of data path connections using the Arrow classes that will be described later in this Chapter.

A predicate name can be created of two ways. The first, if a predicate name has not been created, the user can do it by selecting the option *Create*, then the environment will

prompt him or her to enter the predicate name as well as the number of arguments, this number will match the number of InArrow icons attached to the newly created predicate and it will be stored in the database of predicates **PredicateDataBase** for future reusability. The second way, if a predicate was created, it can be retrieved by dragging the right mouse button to the menu option *Predicate Names*, immediately a menu will pop up whose content is a list of predicate names already created; when the mouse button is released on the desired predicate, the predicate will be displayed along with all its characteristics (See Figure 5.2 below).

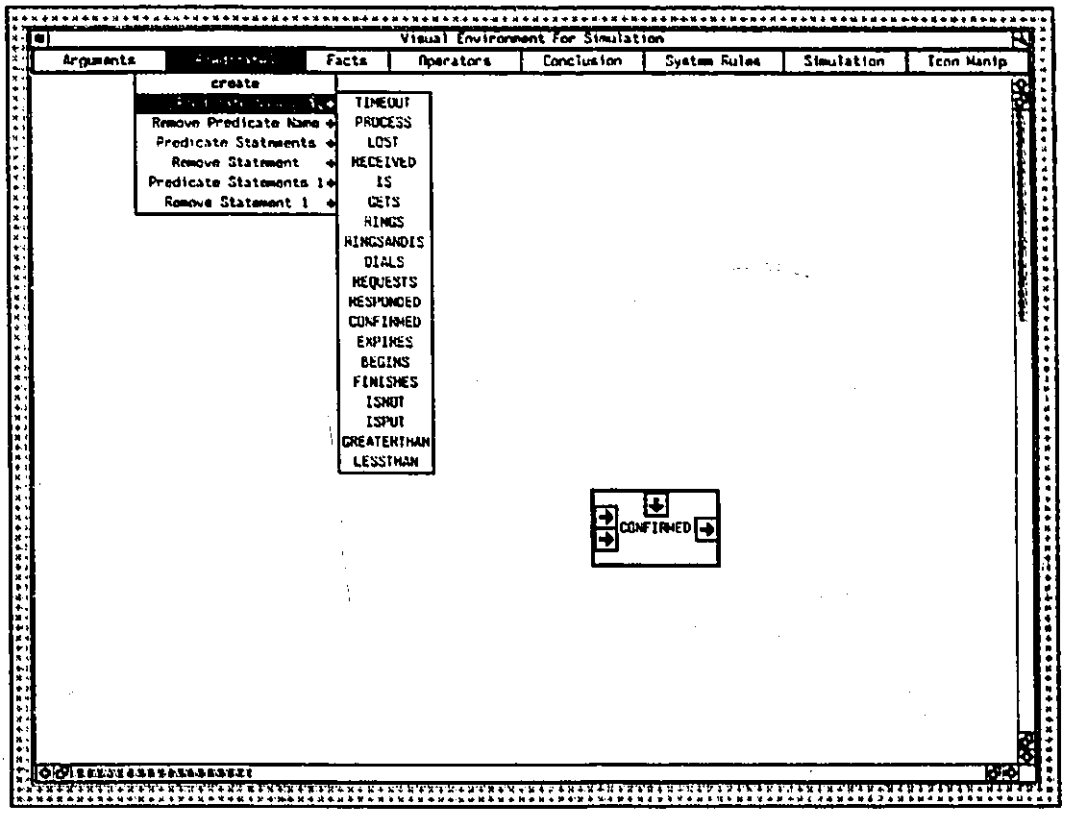

Figure 5.2: The *Predicates* Selection Menu.

The user can delete a predicate name in the same way as it was called from the library, but this time the menu option selected is *Remove Predicate Name* and by dragging the mouse button and releasing it on the predicate name, it is deleted from the database.

An object of this class has attached a number of arrow objects pointing inward (they

belong to the class InArrow), and another arrow object pointing downward which belongs to the class DownArrow for data path connections, and an arrow object pointing outward (it belongs to the class OutArrow) for control path connections. When the arguments are attached to a predicate name through these InArrow class objects, it means that these are the arguments that the predicate name takes. The DownArrow class object helps in connecting the temporal operator to the predicate name. Since the statements formed is going to play certain role within the premise of a rule, the OutArrow class object (control path connection) directs that well-formed statement into a logical operator through the logical operator's InArrow class object.

In the course of this thesis, a statement is a predicate name with all its arguments, temporal operator, and lower and upper bound times attached. All the statements that have been created can be seen using the menu option *Predicate Statements* (See Figure 5.2). The statements will be displayed for possible reuse. In the same way, any of those statements can be deleted using the last menu option.

Each object that belongs to this class presents an independent menu allowing the user to *Quit* provided that no other icons are attached, to *Remove an Icon* that is connected to it, to *Implode* the statement into a single box for working space saving, and to *Save*, if all the connections are established, into the database of statements **StatementDataBase**.

5.2.3 FactCls Class

Objects created under this class behave similarly to those of the PredicateCls class. An object of this class is created by choosing the menu option *Create*, using the third main menu selection, and entering the name as well as the number of arguments as the user is prompted to do so. Once the arguments are attached, the whole expression is ready to be saved in the database of facts **FactDataBase** and also saved in a text file called "facts" whose content is the knowledge of the system (See Figure 5.3). This file is one of the output/input files.

A fact can be either retrieved or deleted be selecting the menu options *Facts* or *Remove Fact* respectively, immediately updating the database of facts and the file "facts" accordingly.

A feature introduced to this menu option is *Edit DataBase* that allows the user to edit a text file with the set of facts that have been created to correct any mistakes or make any changes, if desired, by means of textual manipulation. Any change made by textual manipulation must be reflected in the file "facts.gui" that stores the graphical format of all these facts.

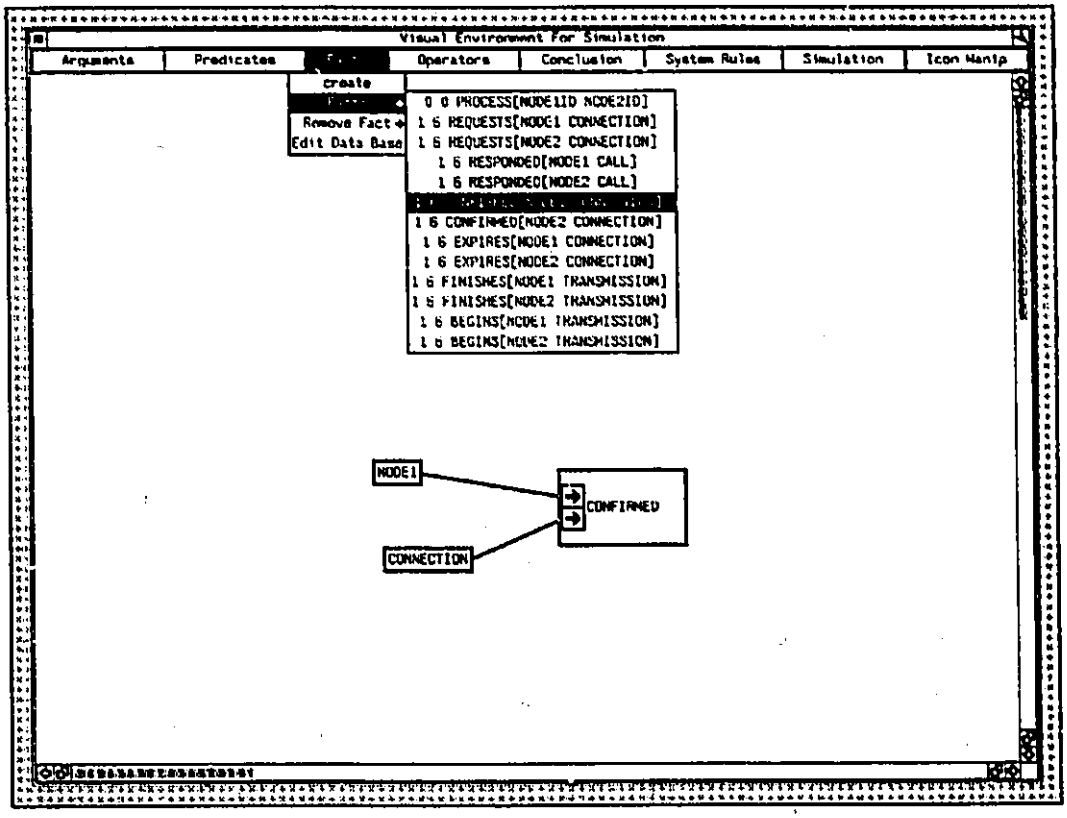

Figure 5.3: The *Facts* Selection Menu.

An object of class FactCls possesses a local menu, this local menu gives the user three options to select from; *Quit*, *Remove*, and *Save* it into the database of facts **FactDataBase.** But when an object of this class still misses one connection the last menu option, *Save*, does not appear to keep the user from, by mistake, saving a fact into the database.

As for the PredicateCls class objects, the FactCls class objects have InArrow class objects attached to represent the assignment of arguments to the fact name.

5.2.4 TemporalCls Class

Objects belonging to the class TemporalCls are the temporal operators used in building the TTLMs. They are created by pressing the left mouse button on the option *Operators* from the main menu and dragging the mouse device to the submenu option *Temporal*, and when the previously defined temporal operators are displayed, release the mouse button on any of them, then the visual environment, interacting with the user, will place an icon on the screen that represents the temporal operator chosen. The top part of Figure 5.4 shows the temporal operators being used. By using the local menu of an object of this class, the user can *Quit* the object wiping it off the screen, or *Remove* a connection.

An icon of this class can only be connected to an object of class PredicateCls through the object whose representation is an arrow pointing downward (class DownArrow). Attempting to connect to another interface class will result in an error message.

Four temporal logic operators have been created and implemented, they are *EVENTUALLY*, *HENCEFORTH*, *NEXT*, and *UNTIL*. They are user-created, so that new temporal operators can be defined and implemented.

Temporal Operators define and provide a consistent way of expressing the behavior of a predicate name in time, they also provide expressiveness to refer to the future. That is, a predicate name is true if and only if the timing constraints set by the temporal operator are met. For example, we use our linear-time temporal logic language to write assertions on behaviors (specifications). For instance, let us say a predicate is true at any time after a time t_1, for this case the temporal operator EVENTUALLY ($\Diamond$) is used to describe the predicate's timing behavior. Another example is the case when a predicate that is true during a time interval, for such situation, the predicate UNTIL ($\mathcal{U}$) is used along with the lower and upper time bounds. In the same way, we can create TTLMs in which the

timing constraint of the predicate implies the use of the temporal operators NEXT (○) and HENCEFORTH (□).

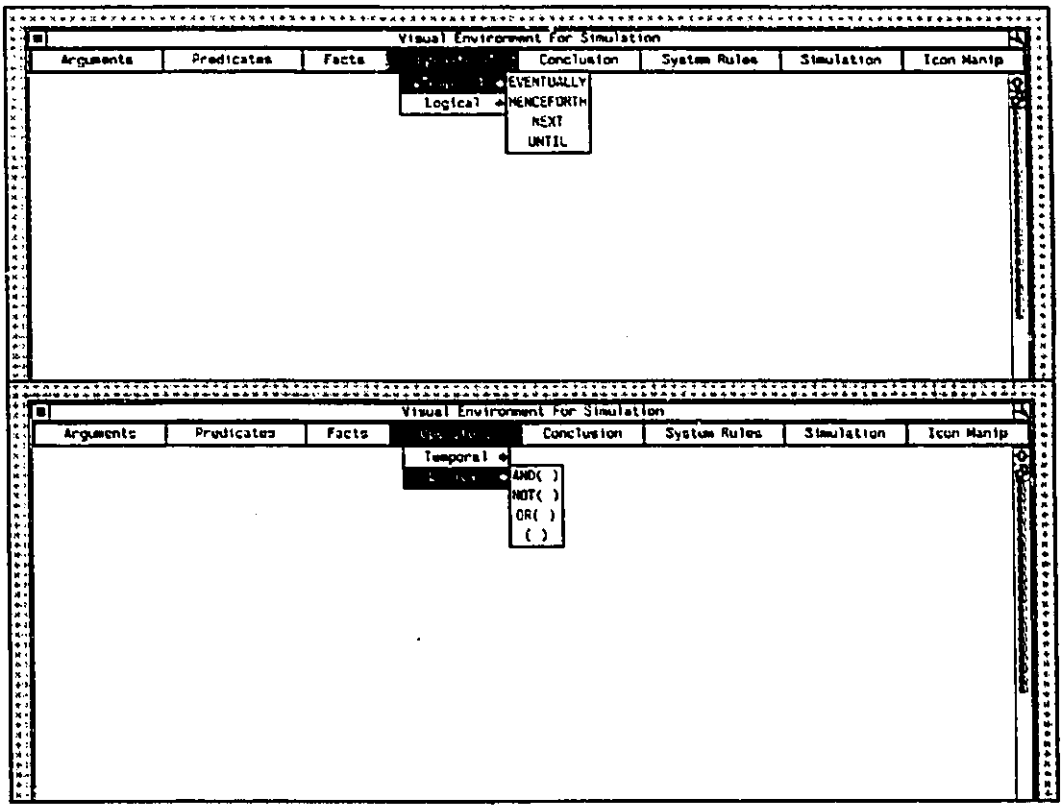

Figure 5.4: The *Operators* Selection Menu: TemporalCls and LogicalCls classes.

5.2.5 LogicalCls Class

The other kind of operators is the logical operators that are objects belonging to the class LogicalCls, they can be seen at the bottom part of Figure 5.4. As for the TemporalCls objects, they are chosen by pressing the left mouse button on the option *Operators* from the main menu and dragging the mouse device to the submenu option *Logical* and releasing it when the cursor is on the desired logical operator. The set of logical operators comprises the logical operators *AND, NOT,* and *OR.* The AND and OR are logical operators that because of their natural definition they need at least two arguments within their premises, so they have been defined to handle virtually an infinite number of arguments, for instance

the following rule can be built:

IF(AND(*argument_1 argument_2* $\cdots$ *argument_n*)THEN(*conclusion*)

When the user attempts to attach more than one statement to the logical operator NOT, this action will be immediately rejected and a warning message will display.

Two kinds of Arrow class objects are attached to an icon of this class, an InArrow object and OutArrow object. The InArrow class object helps in collecting all the statements that will be evaluated under this logical operator as part of the premise of the rule. Since the language described here allows the premise to have more than one logical operator, the OutArrow class object can function in two different ways. It can connect this logical operator to another logical operator, in this case it is being used as a control path connection. Or it can be connected to the DownArrow object of a Conclusion object completing the construction of a rule.

As well as all the other class objects, an object that belongs to this class has its own local menu (See figure 5.5). Understanding the first four menu options is pretty much straightforward. To *Add To RuleBase*, the user should be aware that any icon of this class has the same menu, and that at any time an incomplete rule may be saved. Two warning messages have been added to overcome this problem; when the user tries to save a rule to which a conclusion has not been attached or if in the whole path there is one connection missing, for example, a logical operator OR with just one statement attached.

When a rule is saved into the database of rules, there is a logic involved. First the option *Add To RuleBase* has to be chosen from the logical operator that is the rule of the system, that is, the outer most logical operator of the set of nested logical operators within the rule, or, the one whose control path connection is directed to an object of class Conclusion. Let us assume the user wants to save the rule graphically represented in Figure 5.5, by pressing the mouse button on the icon the local menu is displayed and dragging the pointing device to the *Add To RuleBase* menu option the save operation is realized, but first

the user will be asked where to save it; whether in the **RuleDataBase** (general system), or **BehaviorDataBase** (behavior of the system), or **ControlDataBase** (controller). First the interface writes on a file the string "**IF(**", then takes the name of the logical operator where the save operation started from and writes it right after, so now the partial rule is **IF(OR** ···. There is a method that reads the identification variable of all the input connections attached to this logical operator class object.

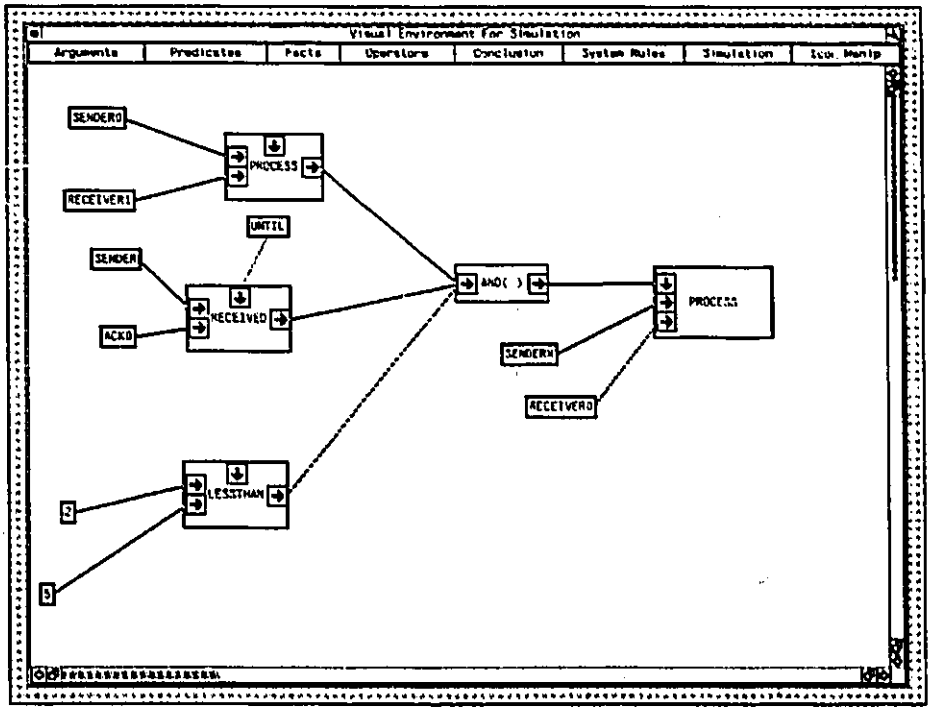

Figure 5.5: Building a Rule Using the GUI.

The class objects attached to a logical operator can be of either PredicateCls class or LogicalCls class. When an object identifier is read and it happens to be of class PredicateCls, this method calls another method to produce a string whose content is the predicate name, arguments, temporal operator, and lower and upper bound times in the following manner:

(lower_bound temporal_operator upper_bound predicate_name [argument_1 argument_2 ··· argument_n]),

coming back to our example the text string copied would be:

(0 0 PROCESS[SENDER0 RECEIVER1]),

that concatenated to the previous string results in:

IF(OR(0 0 PROCESS[SENDER0 RECEIVER1]) ···,

and it returns to the method from which the last method was called.

The next object identifier is the one corresponding to the icon AND, which is of the class LogicalCls, the method takes the logical operator and concatenates it to the string, having the following:

IF(OR(0 0 PROCESS[SENDER0 RECEIVER1])(AND ···,

and the same method is recursively called until all the input connections have been inspected and the strings related to them have been added. So at the end the string has the form:

IF(OR(0 0 PROCESS[SENDER0 RECEIVER1])(AND(1 UNTIL 6 RECEIVED[SENDER ACK1])(0 0 LESSTHAN[2 5]))(0 0 LESSTHAN[2 5])) THEN ···.

Then the conclusion string is built in the same way and is incorporated to the string resulting in:

IF(OR(0 0 PROCESS[SENDER0 RECEIVER1])(AND(1 UNTIL 6 RECEIVED[SENDER ACK1])(0 0 LESSTHAN[2 5]))) THEN(0 0 PROCESS[SENDERX RECEIVER0]);,

and the whole string is dumped into a file.

5.2.6 Conclusion Class

Objects of this class are created by clicking on the fifth option of the main menu bar. Although the objects created under this menu selection appear to be of class Conclusion, they, in fact, belong to the class FactCls. An object of class Conclusion, a hidden class behind the class FactCls, behaves differently to that of the class FactCls to the same methods. They carry information within the definition of the objects that tells their implementation file whether it is an object of class FactCls object or class Conclusion.

In appearance an object of class Conclusion differs from that of the class FactCls in that an arrow pointing downward is attached allowing the control path connection from a logical operator, as described above. Another difference is in the local menu since a Conclusion object is not saved in any database it does not give this option in the set of menu selection. The InArrow objects have the same functions as they were described in Section 5.2.3.

When an object of this class is being created, the user will be asked to textually enter the name and the number of arguments, followed by the lower an upper bound times.

5.2.7 StateCls Class

This class creates the objects for the visual interactive simulation, an animated way to present the outcome of the system under simulation. Other than those instance variables inherited from the class library of Objective-C, an object of this class has three important instance variables: one, to hold the name of the state, another to hold a set of events that might occur and trigger the system toward the next state, and the last one to keep track on the state's local clock. When the right mouse button is pressed on an object of this class a local menu pops up, its content is the set of events allowing for a manual verification of the system as it will be explained later in Section 5.5. In this class, methods for building the reachability graph of both the control and uncontrolled systems are implemented as well as a method to calculate the minimal cost path and display it on the screen.

The instance variable to store the state timing information is used to provide a means of synchronization of different states that run at different clock paces. This local clock is directly related to the global clock and is all the time less or equal than the global clock value. When two processes are communicating, there is the need to synchronize their events such that event occurrence happens at permissible time. For instance an event must wait in a queue until certain state reaches the local clock value to which that event can occur.

Objects of this class allow some connections but neither are they defined as data nor control path connections. Instead, they are used to represent the path followed by the dynamics of the simulation process.

5.2.8 ImplodeCls Class

This class does not contribute with any objects in the creation of TTLMs. But instead, it permits the user to save screen space when such models are getting too big. The method called to perform this task works very similar to that of saving a rule in the database. It takes as the root the object from which the method is being called (an object of either LogicalCls or PredicateCls class) and from there onward it begins saving the iconic diagram into a graphical file, and the diagram disappears from the screen. Meanwhile, an object of class ImplodeCls is created and placed on the screen. This object has the information of the whole iconic diagram that just disappeared.

An iconic diagram is retrieved by choosing the local menu option *Explode* making the diagram appear exactly where it was and quitting the ImplodeCls object that is no longer needed.

An object of this class also allows the user to connect some other allowable temporal logic components maintaining the syntax and semantics of the language.

5.2.9 Connectiveness Classes

This is not really a class, it comprises the set of Arrow classes that are being used to interconnect the TTLM components. These make up the two kinds of connections: Data Path Connections and Control Path Connections, as it was mentioned in Chapter 4.

The Arrow classes used are: the InArrow class, the OutArrow class, and the DownArrow class that depending on the object to which they are attached and the object they are connecting to, help in drawing either data or control path connections. They are transparent objects that serve to symbolically tell the user that objects are to be attached through them.

5.3 Inference Engine

Reasoning about knowledge is a useful conceptual abstraction and an elegant formalism for capturing reasoning that is often given intuitively and operationally or formally but opaquely. Implicit in the term reasoning is the search mechanism which is meant to discover a path through a problem space from an initial configuration to a goal state.

Facts or propositional statements can be thought of as the binary decision with the interrogative (the question mark) removed. Facts are combined through common connectives to give a procedure that solves problems. That is, compound statements (rules) are made up of a collection of elementary propositions where every single proposition has its own truth value and they are used to solve problems through reasoning in an elegant way.

Formal logic usually defines the knowledge. From a logical point of view, this is a declaration of things that are held to be true. It represents the set of all facts that are stated or can be deduced (knowledge of the system).

The inference engine's main task is to reason about the knowledge of the system. To do this, a parser mechanism has been implemented to check for correctness on the written

TTLMs. Then, the data bases of rules and facts are given shape to proceed to reasoning. Before any inference is made, the textual form of the temporal logic models is broken into lexical components and each of them is assigned a code according to its role in the language structure. Once every lexical component is given a code, the temporal model is parsed following some precedence rules. When running the tool to simulate DESs designed in [36] an error writing the specifications means going into an infinite loop since no parser mechanism was implemented to check for grammar correctness.

To begin the dynamical reasoning process consider the view point that the knowledge base and the inference engine, also called the inference mechanism, actually form a closed loop control system as shown below in Figure 5.6

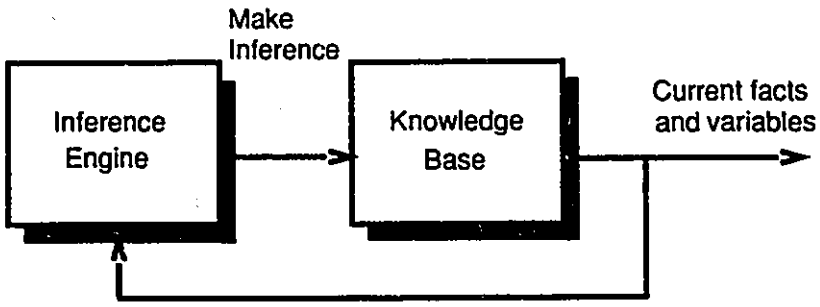

Figure 5.6: Expert System.

The knowledge base (e.g., a rule base) holds the current information about the problem domain. The outputs of the knowledge base, i.e., the current facts and variables, are used by the inference mechanism to decide what inference to make next.

Depending on the inference strategy, different paths of inference will be produced, some of which lead to appropriate conclusions (and appropriate dynamic behavior). As opposed to the inference engine implemented in [36] in which only forward chaining is used, we have used two searching mechanisms: forward chaining and backward chaining (explained in details in previous Chapter). Forward chaining is the strategy of searching forward from the present state or node within the search scene toward the goal. The complementary strategy is to search from the goal back to the initial node or state, and this is called

backward chaining. Let us introduce next all the classes that make possible the application of this two inference mechanisms.

5.3.1 Lexemes Class

An object of class Lexemes has two instance variables to hold a text string and the type that text string is. For instance, a text string can be of type integer, float, predicate, argument, temporal operator, logical operator, etc.

A scan method is implemented to investigate the expressions and to code according to the permissible lexical definitions of the TTLM components. That is, a temporal logic component can be defined, as mentioned above, of type integer, float, predicate, argument, temporal operator, logical operator, etc.

The scan method works as follows, when a component is found, an object of class Lexemes is created and its text string variable is filled with the string that represents the model component and immediately is identified and this identifier stored in the variable that holds the code. Then, this Lexemes class object is stored in an ordered collection of all the components of an expression or TTLM. This ordered collection is, in turn, stored in another ordered collection which is the one that stores all the ordered collections of all the TTLMs or expressions of the file being scanned, just like having a two-dimension ordered collection.

5.3.2 Facts Class

The facts are structures created with this class that have their names, arguments, lower and upper time bounds, and might have temporal operators. They are build from the ordered collection that is returned after the lexical components have been identified or coded.

An instance of class Facts has two instance variables: one to hold the fact name, and

the other, which is an ordered collection of Arguments class objects, to store all the list of arguments, objects of class Arguments, and their respective time bounds. Each Facts class object will be stored in the **FactDataBase**.

The purpose of having a list of objects of class Arguments is to save memory space and to speed up the search for any particular fact, that is, the search stops when the name of the fact is reached then the arguments are compared until the desired fact is found (See Figure 5.7).

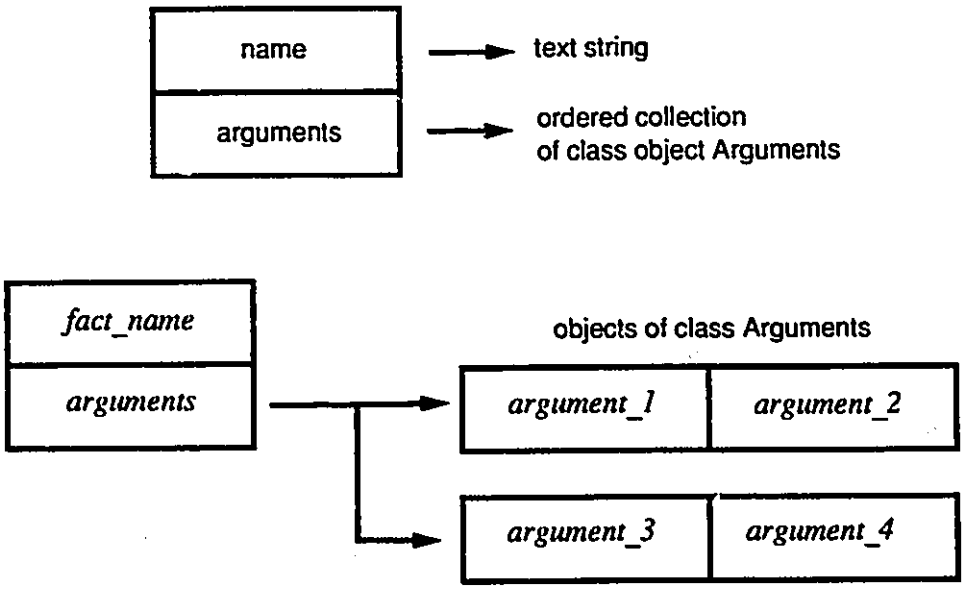

Figure 5.7: Storage Structures of a Fact in the Database.

Assume we have two facts whose names are the same:

fact_name[*argument_1 argument_2*], and *fact_name*[*argument_3 argument_4*],

Figure 5.7 shows how they will be stored in the database of facts. When the fact name being searched is found, the process stops matching the fact name and starts matching the arguments.

The Facts class provides the methods to create the fact structures, to compare two facts and return the boolean value of the search, and some other methods, to stash a fact

into the database of facts **FactDataBase** as well as to delete a fact from the same database.

5.3.3 Rules Class

The rules are built from the ordered collection of lexical components, as for the facts creation, that is returned from the scan method that identifies each TTLM component. They are structures more complex than facts since they include among permissible lexical component logical operators and a conclusion. A rule has the form IF(*premise*)THEN(*conclusion*), in which the premise includes temporal and logical operators, predicate names, and time bounds. An object of this class has two instance variables: one, to hold the premise of the rule and the other the conclusion. The premise is held by a ordered collection of both logical operators and statements and the conclusion is an object of class Facts. For instance, let us say the following rule is being formed:

IF(AND(*statements_1*)(OR(*statements_2*) (*statements_3*))(*statements_4*))
THEN(*conclusion*),

the variable *premise* will hold 6 class objects and the variable *conclusion* will equal the class object that is written after the lexical component THEN, as shown in Figure 5.8. Note that there is no restriction in the number of arguments the logical operators AND, and OR can handle, as far as the evaluation method is concerned these two logical operators can handle an infinite number of statements but in fact, this number is bound by the machine's memory space. The logical operator NOT can only operate on one argument or over the combination of arguments and other logical operators.

Once a rule is built, it might be stored in either of these three databases: **RuleDataBase**, or **BehaviorDataBase**, or **ControlDataBase** depending on from where the creation of rules method has been called.

Methods can be found in this class to create the database of rules, to extract a statement from the premise of the rule, to return a rule whose conclusion matches a Facts class object

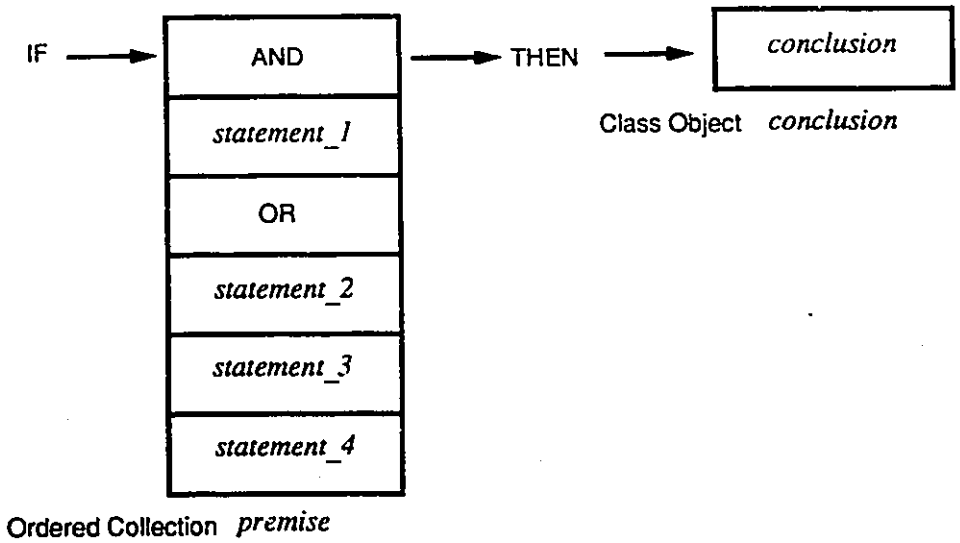

Figure 5.8: Internal Representation of a Rule.

which is the method's parameter passed, etc.

5.3.4 Arguments Class

Objects of this class are created to be stored in the ordered collection called "arguments" of Facts class objects, as seen in Figure 5.7. They have instance variables to store a list of arguments corresponding to their fact names, to store the lower and upper bounds and a cost value for finding the optimal path from a start node to the final node.

Like in the other classes, there are methods defined inside this class. These methods help in telling if the argument lists of two facts are equal taking into account the list of arguments and the time bounds as well.

5.3.5 Parser Class

This class does not have any factory method for the creation of objects of this class. It helps in implementing all the necessary methods to achieve parsing of both the file of facts and the file of rules files.

Once the lexical components are grouped into an ordered collection of Lexemes class objects, they must be parsed for syntax analysis to detect errors if they exist. The parser is sort of LR parser that checks the lexical components following some precedence rules that define the permissible word orders as part of the language (See Figure 5.9). If there exists any errors, an error message will come out on the screen specifying the type of error, the location within the expression being parsed, and the location number of the expression in the file as well as the file name.

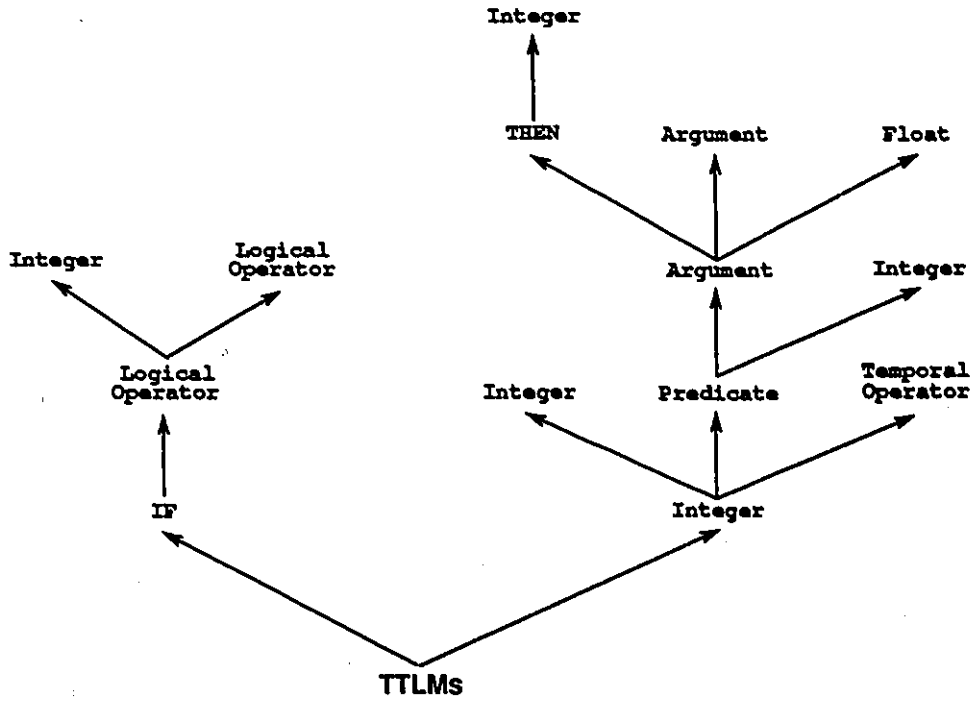

Figure 5.9: Parser's Permissible Sequences.

When parsing a fact or a rule the same method is used. The parser knows whether the expression is a fact or a rule because it inspects the first element of the ordered collection of lexical component and if it is the word **IF** that means the expression is a rule, otherwise it is a fact.

The rules of permissible sequences (Figure 5.9) work as follows: assuming that the first lexical component of the TTLM is the word IF, the next lexical component must be a Logical Operator. From a Logical Operator lexical component, there are two possible

paths that the next lexical component is an Integer or a Logical Operator again. If it is an Integer three options are possible. The next lexical component can be an Integer, a Predicate name, or a Temporal Operator, and so on and so forth until the the last lexical component of the TTLM has been parsed.

5.3.6 Search Class

Choosing a clever search mechanism can be critical to the success of a rule-based system. Sometimes certain aspects of a problem can be best handled via forward chaining and other aspects by backward chaining. The kind of rules implemented allows the inference engine to use a combination of both search mechanisms, forward and backward chaining.

Two inference mechanisms are implemented in this class, for them, two methods are being used: one to forward search and the other to backward search.

Forward chaining is the strategy of searching forward from the present state or node within the search scene toward the goal. The method to perform forward chaining takes a parameter, a goal (an object of class Facts). This goal, in turn, is sent by this method to another method in the class Facts to inspect its existence in the database **FactDataBase** returning the boolean value. This inspection is done from top to bottom, and whenever the goal is found it returns a positive boolean value to the search controller method. If the bottom is reached, and the boolean value is still negative, the search controller calls the backward chaining method having as parameter the same goal.

The backward chaining strategy is to search from the goal back to the initial node or state. When the backward chaining method is called the parameter passed through is taken and sent to a method in the class Rules. This goal is compared to the conclusions of all the rules in the database of rules. All these rules whose conclusion matches the goal, are stored in a temporal ordered collection. Then, the first element of this temporal ordered collection is taken and sent to the search controller for forward chaining search.

Whenever a rule is fired true, its conclusion is stashed in the database of facts **Fact-**

DataBase for further reasoning.

5.3.7 Evaluation Class

The class Evaluation uses four methods to achieve rule evaluation. There is a method to control the evaluation process and three more methods to evaluate the logical operators.

First, a message is sent to the evaluation controller method with the premise as a parameter and a global counter (pointer), that points to the element of the premise, is set to zero. Whenever a left parenthesis is found a local counter is incremented and is decremented when a right one is found.

When the message gets to the evaluation controller method, the element of premise pointed be the counter pointer is analyzed. If it is a left parenthesis, a parenthesis counter is increased. If it is a right parenthesis, the same counter is decreased. If it is a statement, it is sent to the search controller for reasoning. If it is a logical operator the evaluation process is sent to a logical operator evaluation method. The process of taking the element of the premise pointed by pointer and analyze it is done until the parenthesis counter is zero again.

As there is a logical operator method for evaluating the AND operator, there is a method for evaluating the OR operator, and another one for the NOT operator which work very similarly with slight differences.

Once the evaluation process is in any logical operator method, a local counter of parenthesis is set to one. The element pointed by pointer is taken and analyzed, the choices are the same as those described in the paragraph above, until the parenthesis counter reaches the value zero.

Let us show how the evaluation is performed by means of an example. Consider the rule:

IF(AND(*statements_1*)(OR(*statements_2*)(*statements_3*))(NOT(*statements_4*)))

THEN(*conclusion*),

whose memory structure is shown in Figure 5.10. The first parenthesis is analyzed and the counter increased, then the logical operator AND makes the evaluation controller sent a message to the logical operator AND evaluation method. In this method, another local parenthesis counter is set to one, and the next element analyzed. The next element is the logical operator OR, then the message is sent to the logical operator OR evaluation method in which *statements_2* and *statements_3* are evaluated and when the parenthesis counter reaches zero the result of this evaluation is returned to the logical operator AND evaluation method.

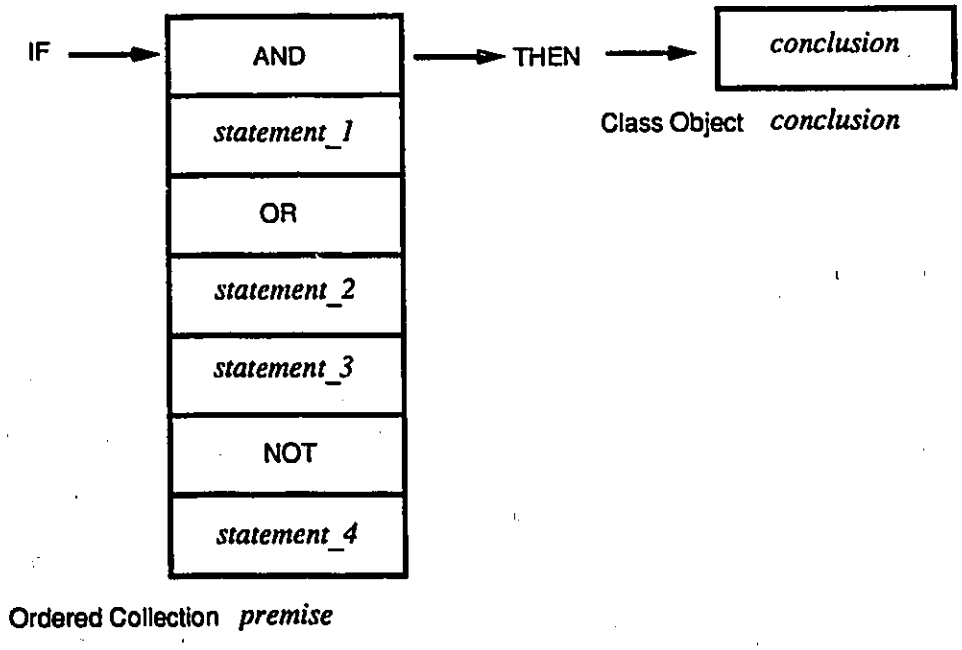

Figure 5.10: Rule Evaluation Process.

While back in the logical operator AND evaluation method, the evaluation process keeps on analyzing the elements of the premise, this time the element is the logical operator NOT, then the message is sent to the method corresponding to this logical operator in which *statements_4* is evaluated and the result of this evaluation returned accordingly to, once again, the logical operator AND evaluation method. From this point on, the evaluation method keeps updating the parenthesis counter until it is zero, and when the counter is

zero it returns the evaluation outcome to the evaluation controller.

5.4 Visual Simulation

This block offers the facility of displaying the dynamic behavior of the system components on the screen. This facility is user-interactive and allows the system to react to user actions such as simulation halts, request of additional information, etc. To achieve displaying the system behavior on the screen, three classes interact with one another, they are the classes StateCls, States and Events (See Figure 5.11).

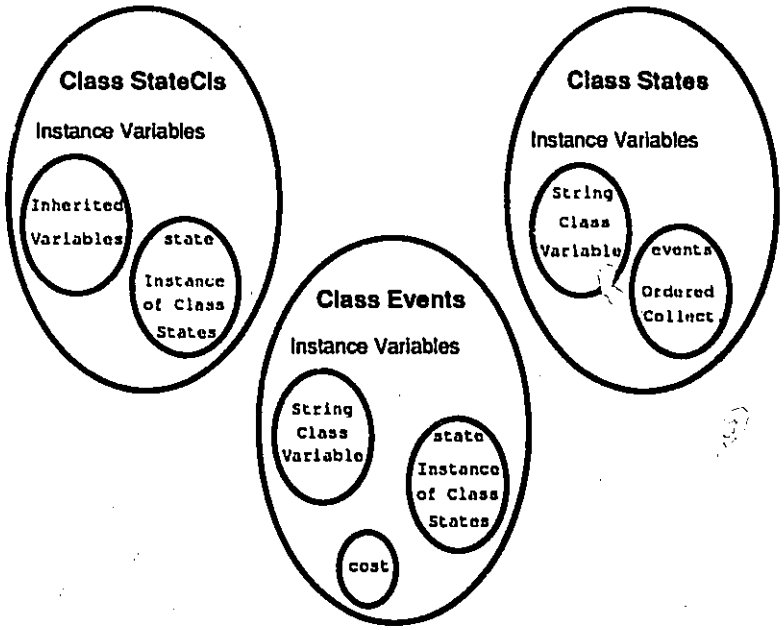

Figure 5.11: Interacting Classes To Achieve Dynamic Simulation.

The class StateCls, described in Section 5.3.7, is the interface class between the graphical user interface and the information about the different states of the system. An object of this class has all the instance variables that allow it to be displayed on the screen, react to mouse button events, etc. These instance variables are inherited from the hierarchy class that the object-oriented programming language Objective-C offers. Apart from them, it has another instance variable which is an object of class States in which all the information

about any particular state is kept.

5.4.1 States Class

The information about a state is stored in objects of class States. The instance variables of an object that belongs to this class are a String class object to hold the state name, and an ordered collection of events to store all the events related to that state which will lead, eventually, the simulation process to a next state. These objects stored in that ordered collection are objects of class Events which is described next.

This class provides some methods that are helpful when the simulation is run, some of these methods help in building the states, extracting an event from the ordered collection of events, given the state finding the event in a rule from the database of rules, editing a file of both the general system and controller behaviors, etc.

Figure 5.12 refers to the AB Protocol (ABP) example. This protocol is simpler than others and it was chosen to ease the understanding of this visual environment. The ABP is used to coordinate the flow of messages between two processes, a *Sender* process on one node and a *Receiver* process on another, in a distributed network. The *Sender* transmits a packet numbered 0 and it will not transmit a packet numbered 1 until it receives, an acknowledgement that the packet was received correctly by the *Receiver*. The message is placed in a packet with one bit sequence number. The service supported by the ABP is to deliver packets enqueued at the *Sender*'s site in a first-in first-out order to the *Receiver*'s site. The *Sender* and *Receiver* comprise the protocol layer; they communicate with each other via the lower-layer transmission services.

5.4.2 Events Class

The class Events has a variety of instance variables among which the most important are: an instance variable to store the event which, in fact, is an object of class Facts (described in previous section), another instance variable to store the next states to which

the process would go in the case that this event has occurred, this variable is an instance of class States, and an instance to store the cost used to calculate the optimal path.

There are several methods implemented in this class. One of the most important ones is the one that finds the event in an ordered collection of events that matches a parameter being passed and returns its corresponding next state to keep the simulation going. Another important method is the one that creates the database of all the events in the system and assigns a cost value to each one of them.

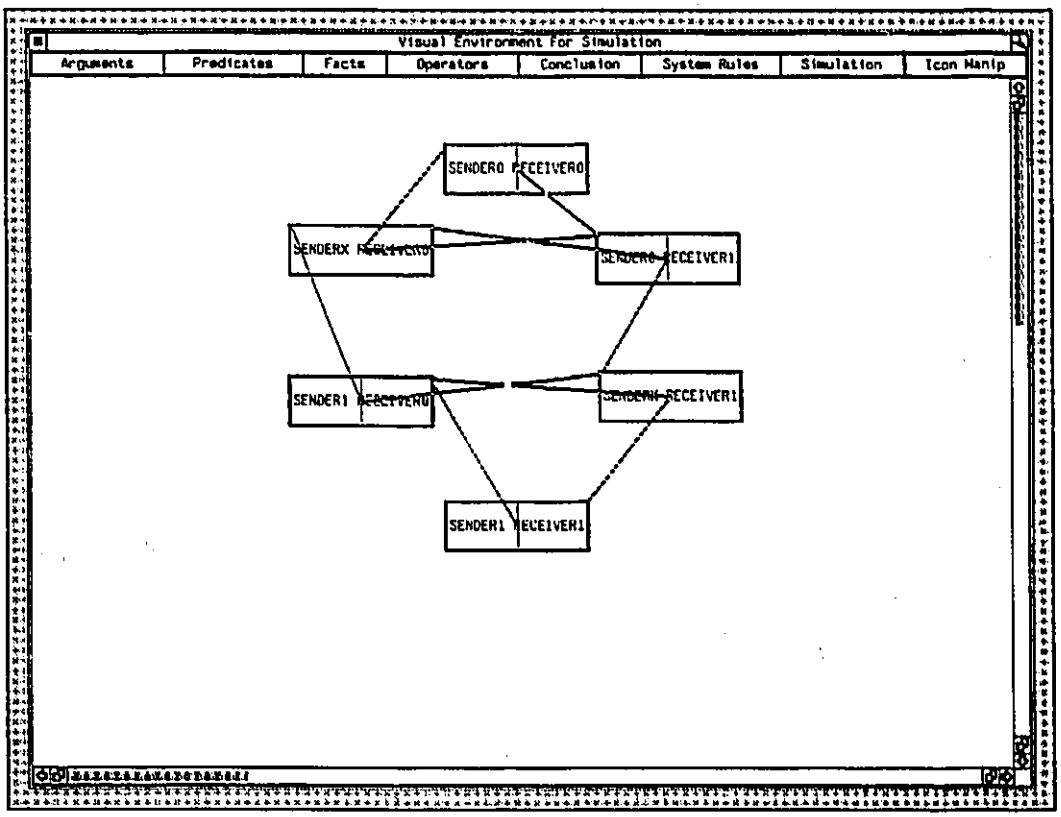

Figure 5.12: Reachability Graph of the System Being Simulated: AB Protocol (ABP).

5.4.3 Simulator Class

This class has no factory method for object creation. It provides the main routines to perform the simulation. These routines carry out the tasks of calling the respective methods to create the states of both the uncontrolled and the controlled systems and save

them in dynamical databases and create the database of events.

All the databases created in the main routines will allow the user to place physically the states on the screen of both systems (uncontrolled and controlled) as well as create and show the optimal path. In Chapter 6, a detailed explanation of how the above tasks are accomplished will be given.

5.5 System Facilities

Although they do not represent any object creation, we would like to mention the facilities found in the last three options from the main bar menu: *System Rules*, *Simulation*, and *Icon Manipulation*.

5.5.1 System Rules

No class objects are created through the menu selection, *System Rules*. With this menu selection the user can look at the whole set of system rules which are the rules of the general system, the rules that set the behavior of the system, and the rules upon which the system in being controlled. Figure 5.13 shows the three menu selection and their submenu options which are basically the same differing one another from the database from which the information is being read. For instance to build the ad hoc menus to show the rules for the general system, for the system behavior, and for the controller, the databases **RuleDataBase, BehaviorDataBase**, and **ControlDataBase** are respectively read.

An ad hoc menu is a menu that is updated and built in real time, once the user requests such a menu, the program builds the logical menu by creating an ordered collection with all the menu options available at that time and displays on the screen the physical menu. If any changes have occurred and the menu is called again those changes will be reflected. This allows the user to add and delete information and have the most updated database to work with.

The first submenu option allows the user to display on the screen any rule chosen by the user. The rule is read from a graphical file whose content is the information regarding the iconic diagram such as predicate names, arguments, temporal and logical operators, etc., as well as some physical information such as coordinates, this is where the icons were located on the screen when saved.

The following submenu option *Remove a Rule* takes the chosen rule and deletes it from the graphical file and from the database maintaining these two files consistent with one another. The third submenu option allows the user to display and edit a file on the screen and look the textual form of the TTLM in our language.

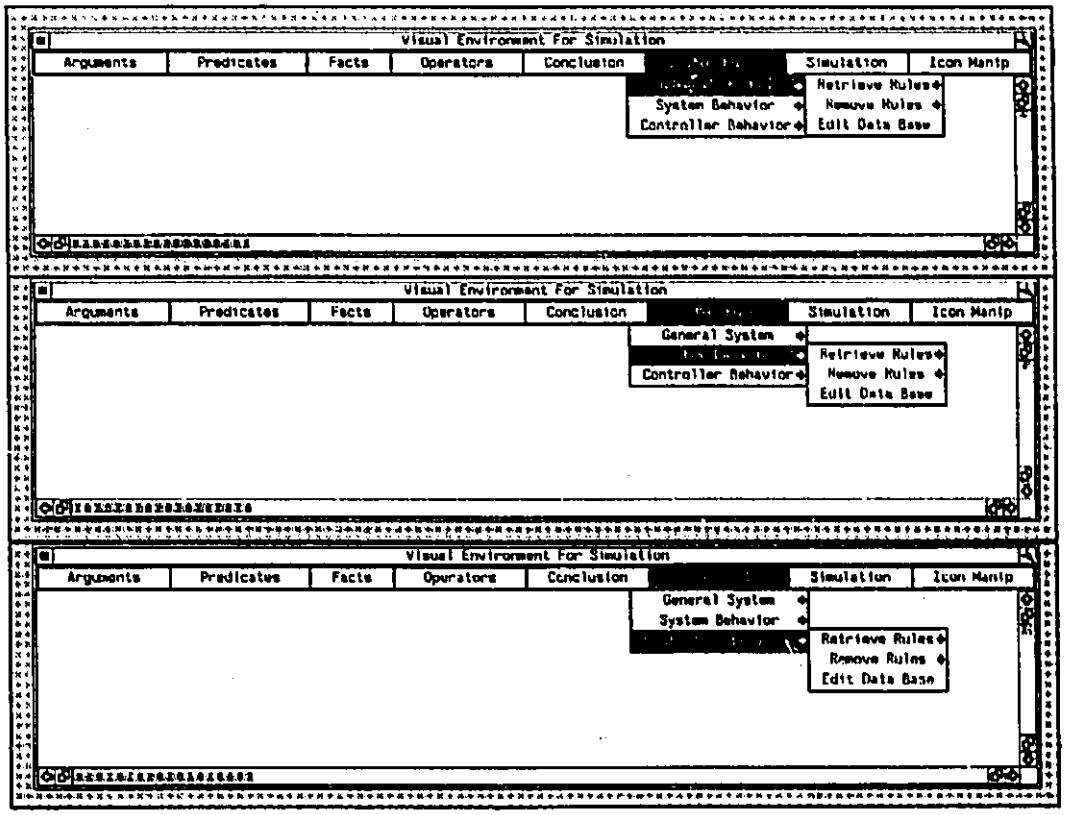

Figure 5.13: The *SystemRules* Selection Menu.

From this menu selection, *System Rules*, the user can look at the file whose textual contents are the TTLMs for the general system, for the system behavior, and for the controller following our syntax and semantics. These files, later on, are going to be evaluated and

from these evaluations two sets of states and their corresponding events will be generated. Meanwhile, two reachability graphs are built; the reachability graph of the uncontrolled system and the reachability graph for the controlled system. This will be described next.

5.5.2 Simulation

Through the *Simulation* selection menu the user runs the simulation process for DESs. By first selecting the *Simulating* option (See Figure 5.14). It starts by calling the inference engine block which reasons about the knowledge of the system based on the rules and facts previously created and stored using the GPE. The states of the system are created by evaluating the rules of the general system from the file "rules" against the database of facts stored in the file "facts". Another file of rules, the system behavior rules, whose content sets the behavior of the system, tells which states are not reachable and which events cannot occur at any particular state. Following that, the file "ControlRules" that contains the rules of the controller that generate a set of states that is a subset of the set of states of the general system in which the number of reachable states might be less and so is, probably, the number of events.

The *Uncontrolled System* option, (middle part of Figure 5.14), allows the user to materialize the states and interactively place them on the screen. They are created as they are being placed on the screen until the last one is created, according to the information previously inferred. All the possible paths are established from one state to another (the reachability graph). Each state created belongs to the class StateCls which among its instances variables there is one variable called "events" (ordered collection) that contains a set of events that might occur at that state and the corresponding next state provided that the event has occurred and the time constraints have been met. The reachability graph can be also generated in a textual form by choosing the submenu option *Edit Graph*.

The *Controlled System* option allows the user to do two different things: first, to place the reachability graph of the controlled system on the screen and second to edit a file with

the textual interpretation of the reachability graph. When placing the reachability graph, the method called first sees if the reachability graph of the uncontrolled system is already on the screen, if yes, all the states of the reachability graph of the controlled system are put over their corresponding states of the reachability graph of the uncontrolled system. If the reachability graph that corresponds to the uncontrolled system is not on the screen, all the states of the controlled system are placed, interacting with the user, on the screen.

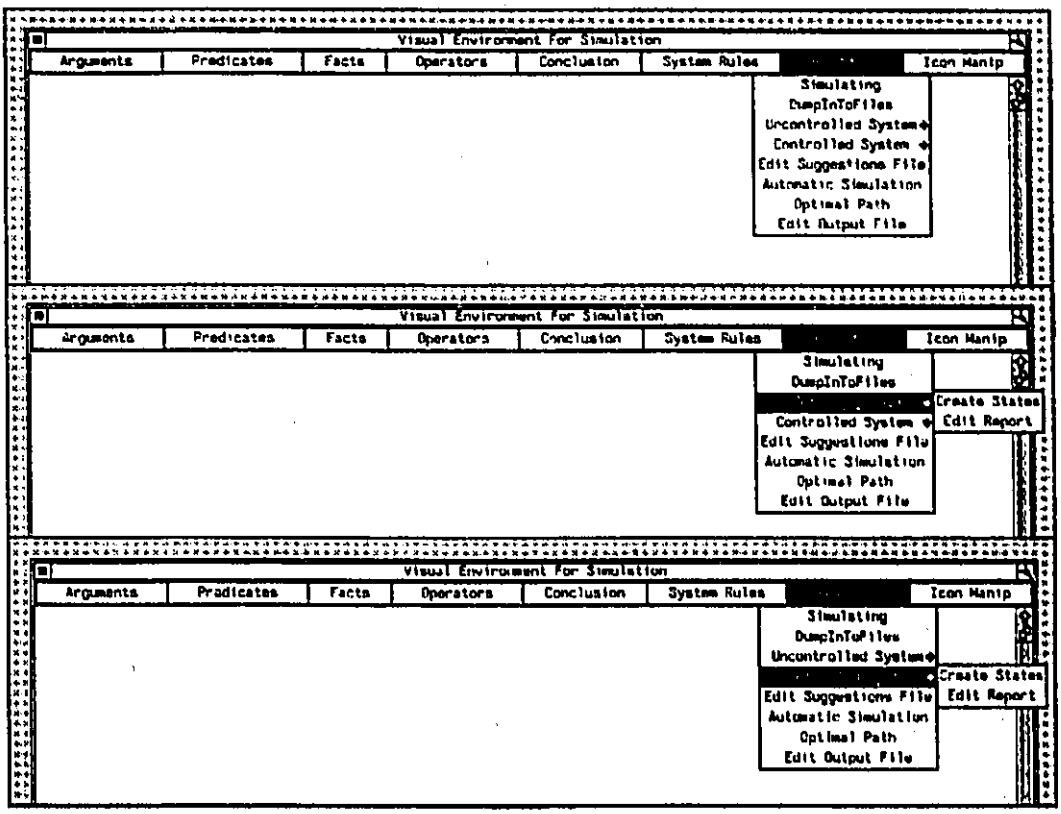

Figure 5.14: The *Simulation* Selection Menu.

A monitor described in the previous Chapter has been included to keep track on those unreachable states and events that cannot occur while the process is at any states. This monitor gives some recommendations that help the user in building the controller. Such suggestions can be seen in a file that is edited by selecting the option *Edit Suggestions File.*

The next two options (*Automatic Simulation* and *Optimal Path*) permit to randomly generate an event and according to that event the process is taken the corresponding next

state and investigate every state of the reachability graph coming up with the optimal path, respectively.

5.5.3 Icon Manipulation

The last option from the main menu bar, *Icon Manipulation*, is to perform maintenance tasks such as *Clear* the screen, *Save* an iconic diagram into a graphical file and *Retrieve* those previously stored graphical files and plot them on the screen.

5.6 Conclusion

In this chapter we have presented most of the classes used to build the architecture of the GPE which comprises three blocks: the Graphical User Interface (GUI), the Inference Engine, and the Interactive Simulation.

The GPE allows the user both to create the knowledge bases of the system (general database of the system, database of rules of the system behavior, and database of rules of the controller, as well as some initial conditions and database of facts) and to run the simulation interactively. Some other functions of minor importance are also handled through the GPE such as edit a database file, edit the textual representation of a reachability graph of the system being simulated, etc.

A combination of two search mechanism, forward and backward chaining, is used as means of improving the inference process when firing a rule. Reasoning can be done forward from the goal to the conclusion and backward from the conclusion to the goal.

The interactive simulation is performed in, virtually, any discrete event systems provided that their behaviors have been modelled through TTLMs and saved in the respective databases. It allows for displaying of both the uncontrolled system and the controlled system as well as the optimal path to go from an initial state to a final state that are previously specified. A way of measuring time is implemented to run simulations in real time.

The display facility of the dynamics of the system is achieved. This feature significantly increases the efficiency of the user. It enhances model verification, validation, and testing.

Chapter 6

Examples of Simulation Models of DESs Using the GPE

The specification of real-time DESs are written using the TTL language defined in previous chapters. This chapter describes by means of two examples how the TTLMs built using the GPE can be used to model DESs, such examples are the packet-switched communication network and the telephone switching network. Through these two examples a real-time simulation of DESs using timed temporal logic is performed and the features of the GPE are also shown.

6.1 Introduction

In control theory, the complete system is often divided into two parts called the *plant* and the *controller*. The plant is that part of the system to be controlled. The open-loop behavior of the plant, or uncontrolled system, is usually unsatisfactory in some respect. It is the task of the controller to ensure that unsatisfactory behavior in the plan is eliminated.

Lin in [36] states that the plant of a DES can be represented by a TLM plant which is composed of a number of TLMs representing the plant processes. The specification of a

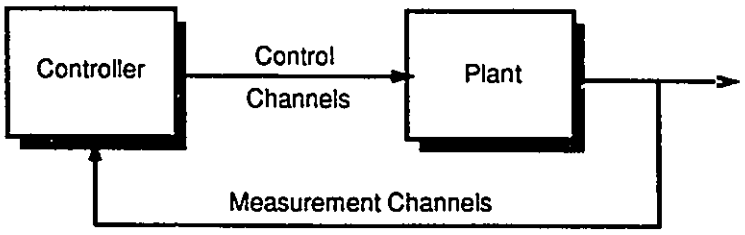

Figure 6.1: Closed-Loop System: Plant and Controller.

plant is a set of possible events that can occur in the system. The description of the required plant behavior is commonly a restriction on the reachability set of the system such that the unsatisfactory states may not be reached. The controller has the same set of events as the plant and its task is to ensure that the unsatisfactory behavior of the plant is eliminated.

Lin in [36] also shows a procedure for controller synthesis for realizing the specified closed-loop behavior. Here, it is proposed that the two systems (plant and controller) operate concurrently and that control actions are achieved by the synchronization of events in the plant with the events in the controller. Therefore, the behavior of the initially uncontrolled plant is restricted since certain selected events are prevented from occurring unless the controller is also in the state which allows the corresponding synchronized event to occur. In other words, the plant generates the events that have to be recognized by the controller to ensure that the unsatisfactory behavior is removed of the plant.

In aiming toward an optimization for DESs in a temporal logic framework, our motivation is to design a controller to generate a sequence of events to drive the system from an initial state to a final state along a trajectory that has the minimum cost.

A simulator of DESs should include a mechanism to display the dynamical behavior of the system. This display feature should also allow user-interaction with the running program. The display facility would help the user observe the execution of the system facilitating the verification of the plant as well as the controller.

What follows are two examples of DESs. In the first example the specifications of a

packet-switched communication network are defined using the TTL language introduced in Chapter 2, a real-time simulation is run to observe the timing behavior of the communication network. In the second example, the telephone system, the features implemented in the GPE are shown.

6.2 Example 1: Packet-Switched Communication Network

The packet-switched communication network guides and regulates the flow of packets reducing the travel times of packets through the network and increasing the amount of traffic the network can carry. The packet-switched network is described in [24, 36]. Figure 6.2 shows the state-transition diagram of the call establishment phase.

The system has four states: **ID** that means idle, or there is no connection established; **WC** stands for waiting for a call connected packet; **CR** stands for connection established and ready for data transfer; and **DT** that means data transmission.

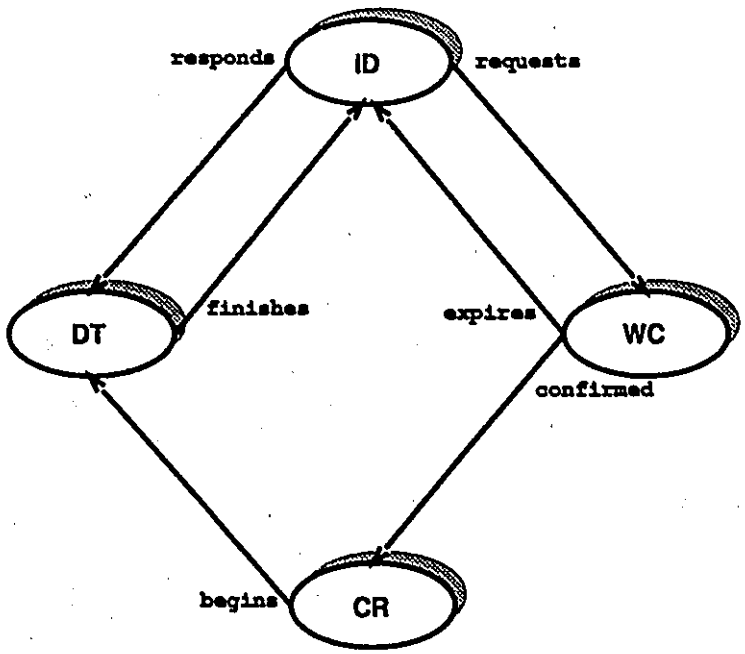

Figure 6.2: State-Transition Diagram of the Call Established Phase.

The system also has six events. There are two events at state **ID** that can occur:

requests means that a connection is being requested; and *responds*, call response. At state **WC** the events *confirmed* and *expires* can occur, they stand for connection confirmed and call connected timer expires, respectively. At state **CR** only one event can occur: *begins* which stands for data transmission begins. At state **DT** the only event that can occur is *finishes* to signal the end of data transmission.

The simulation is performed for a system of two packet switched nodes (PSNs) that have the same state-transition characteristics as described above. The specification of the system which is a set of TTLMs will be given by means of interconnecting icons through the GPE.

For the purpose of the example, to establish a difference between the two nodes, the name of the states in the state-transition diagram have been modified. For instance, when the PSN number one is idle, it is named NODE1ID; when it is waiting for a call connected packet, NODE1WC; NODE1CR and NODE1DT for the two remaining cases, respectively.

6.2.1 Plant Specifications

A specification of a process is a predicate that describes its behavior. The plant of a DES is composed of a set of temporal logic models that represent the plant processes [36].

The state of the packet-switching communication network protocol is $s = (x, y)$, where x represents the state of the PSN number 1 and y the state of the PSN number 2. Following is a set of formulas of the specification for the packet-switching communication network that have been written using the time temporal logic language described previously in Chapter 2:

Dynamics

$$\Box[(ID(x) \wedge \mathit{requests}(x)) \wedge \mathcal{U}(0 \leq \kappa \leq 3) \Longrightarrow WC \bigcirc (x)] \qquad (6.2.P1)$$

$$\Box[(ID(x) \wedge \mathit{responds}(x)) \wedge \mathcal{U}(0 \leq \kappa \leq 13) \Longrightarrow DT \bigcirc (x)] \qquad (6.2.P2)$$

$$\Box[(WC(x) \wedge \mathit{confirmed}(x)) \wedge \mathcal{U}(4 \leq \kappa \leq 7) \Longrightarrow CR \bigcirc (x)] \qquad (6.2.P3)$$

$$\Box[(WC(x) \wedge expires(x)) \wedge \mathcal{U}(13 \leq \kappa \leq \infty) \Longrightarrow ID \bigcirc (x)] \qquad (6.2.P4)$$

$$\Box[(CR(x) \wedge begins(x)) \wedge \mathcal{U}(8 \leq \kappa \leq 10) \Longrightarrow DT \bigcirc (x)] \qquad (6.2.P5)$$

$$\Box[(DT(x) \wedge finishes(x)) \wedge \mathcal{U}(11 \leq \kappa \leq 13) \Longrightarrow ID \bigcirc (x)] \qquad (6.2.P6)$$

$$\Box[(ID(y) \wedge requests(y)) \wedge \mathcal{U}(0 \leq \kappa \leq 3) \Longrightarrow WC(\bigcirc y)] \qquad (6.2.P7)$$

$$\Box[(ID(y) \wedge responds(y)) \wedge \mathcal{U}(0 \leq \kappa \leq 13) \Longrightarrow DT \bigcirc (y)] \qquad (6.2.P8)$$

$$\Box[(WC(y) \wedge confirmed(y)) \wedge \mathcal{U}(4 \leq \kappa \leq 7) \Longrightarrow CR \bigcirc (y)] \qquad (6.2.P9)$$

$$\Box[(WC(y) \wedge expires(y)) \wedge \mathcal{U}(13 \leq \kappa \leq \infty) \Longrightarrow ID \bigcirc (y)] \qquad (6.2.P10)$$

$$\Box[(CR(y) \wedge begins(y)) \wedge \mathcal{U}(8 \leq \kappa \leq 10) \Longrightarrow DT \bigcirc (y)] \qquad (6.2.P11)$$

$$\Box[(DT(y) \wedge finishes(y)) \wedge \mathcal{U}(11 \leq \kappa \leq 13) \Longrightarrow ID \bigcirc (y)] \qquad (6.2.P12)$$

Formula 6.2.P1 describes the effect of that PSN number 1 requests connection to the switching network, thus implying the state change from being idle (ID) to waiting to be connected (WC). This state change is subject to the timing constraint imposed by the clock variable κ that must be in between the time bounds. Similarly, the formulas (6.2.P2) to (6.2.P12) describe the state transitions due to the occurrences of the corresponding real-time events.

The system specifications (formulas 6.2.P1 to 6.2.P12) are taken by the inference engine for reasoning. The outcome is the reachability graph with the enabled events which will be shown later in this Chapter.

Once the state-transition diagram is understood, all the TTLM components must be identified in order to create the necessary objects to construct the TTLMs. Names of predicates, arguments, logical operators, temporal operators, and time bounds need to be identified.

As mentioned before, the system consists of two packet switched nodes (PSNs). Each state at which each PSN is, is considered an argument, for instance, **WC1** means that

PSN1 is waiting for a call connected packet, **DT2** means that PSN2 is transmitting data, so on and so forth. Therefore, each event of the system is taken as a predicate name.

The GPE allows the user to write the state-transitions of the states by interconnecting icons. For instance, to write the state-transition of state **ID** when the system formed by two PSNs is at state **NODE1ID NODE2ID** (PSN1 and PSN2 are idle) and the event PNS1 is requesting connection. To write the state of the PSNs predicate name PROCESS is introduced. To write the fact that PSN1 is requesting connection, the event *requests* might be thought of as being a predicate name such that REQUESTS is created as a predicate, and NODE1 and CONNECTION being the arguments for this predicate name. Now, this expression is bound in time, that is, the transition can only occur during a fixed time period. The predicate name LESSTHAN is created to represent this time binding, it takes two arguments that are the lower and upper bounds, t_l and t_u respectively, and it simply means that the expression is valid if and only if the value of the global clock is greater than the lower bound and less than the upper bound ($t_l \leq \kappa \leq t_u$). And the conclusion will be the next state to which the system will go, that is **NODE1WC NODE2ID**. Note that for this timed temporal logic model the PSN1 is the only one affected by the transition, so the PSN2 remains at the same state. Eight ArgumentCls class objects, three PredicateCls class objects, one LogicalCls class object, and one Conclusion class object.

First the predicate names that do not exist in the database of predicate names are created. A predicate name is created by choosing the submenu option *Create*, see Figure 6.3. Immediately, the user will be asked to enter the name, PROCESS, followed by the number of arguments (two) that is the number of ArgumentCls class objects this predicate can take. Then, this PredicateCls class object is created, saved in a database for future reuse, and placed on the screen. Using the same procedure, the predicates REQUESTS and LESSTHAN are created.

To form the premise of this TTLM six arguments need to be created. The user creates them by clicking on the bar menu option *Arguments*, the left most one. Once created, this object has a local menu option that allows the user to change the name. Six Argument-

Cls class objects are created and changed their names to NODE1ID, NODE2ID, NODE1, CONNECTION, 0, and 3. The first two arguments are connected to the predicate name PROCESS, the following two to the predicate REQUESTS, and the remaining two to the predicate LESSTHAN, through their data path connections or InArrow class objects.

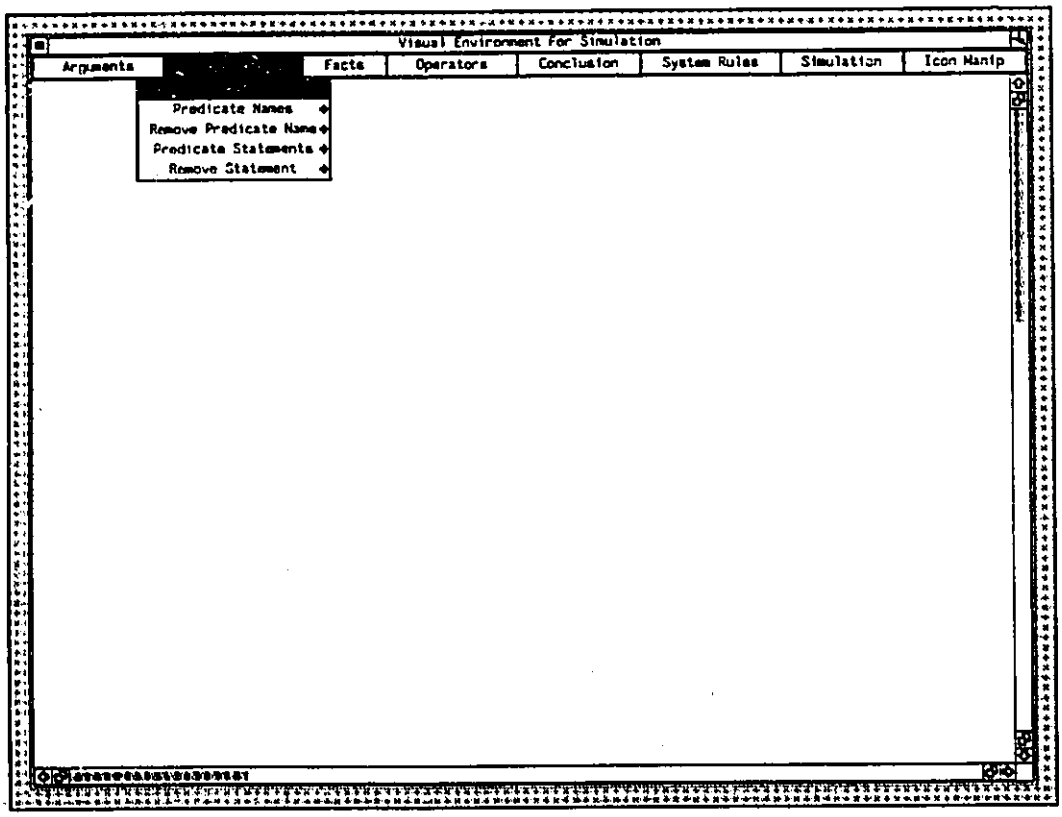

Figure 6.3: PredicateCls Class Object Creation.

Figure 5.5 shows the bar menu option to create the temporal and logical operators. Dragging the mouse device with its right button depressed, the user selects the logical operator AND, and the temporal operator UNTIL.

The temporal operator is connected through the DownArrow class object to the predicate REQUESTS. The predicates are connected through control path connections to the logical operator AND. From the logical operator's local menu, the rule is saved into the database of rules **RuleDataBase**, provided that the graphical TTLM is saved, the conclusion has been constructed and attached to the logical operator through its OutArrow

class object, otherwise, the GPE will not allow the user to save it and will display an error message.

A Conclusion class object is created when the user selects from the bar menu the option *Conclusion*. The user is prompted to enter the name of the conclusion which, in facts, is an object of class FactCls followed by the number of arguments and lower and upper time bounds. The user enters PROCESS as the conclusion name, then enters the decimal digit two, and the time bounds are both zero. The names of the two arguments are changed to NODE1WC and NODE2ID and attached to the conclusion. Once created, the conclusion is attached to the logical operator AND. Figure 6.4 shows the appearance of the iconic diagram of the TTLM just built.

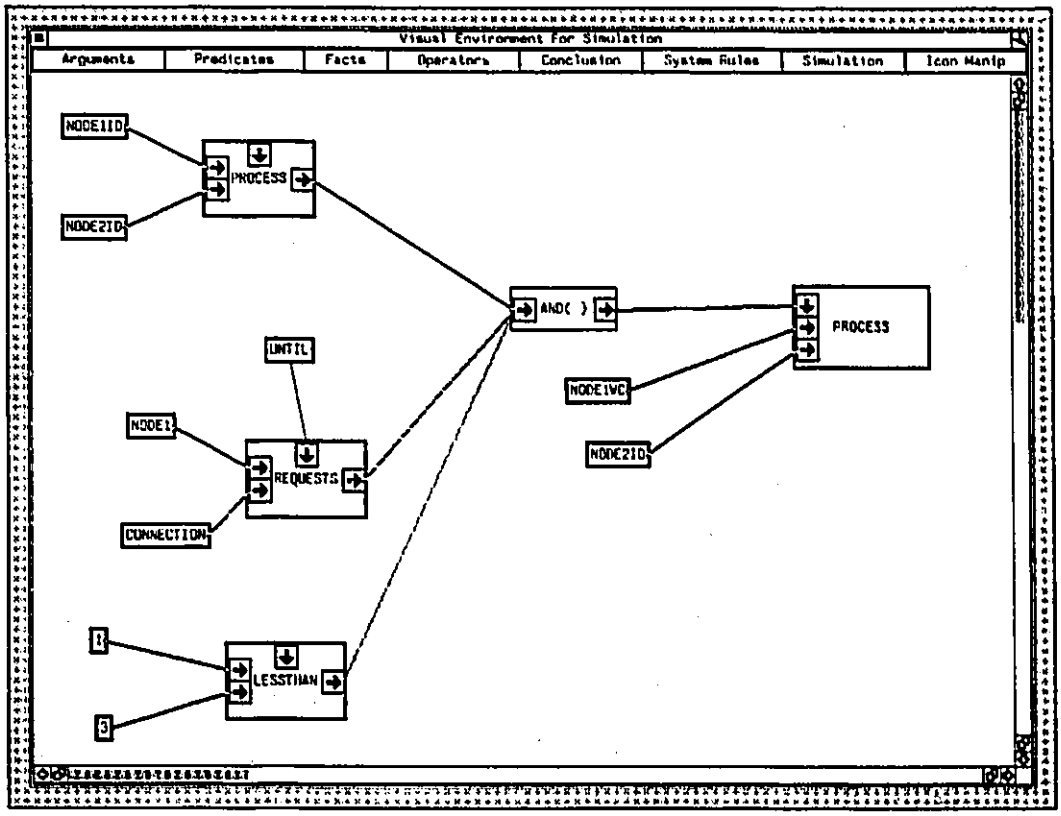

Figure 6.4: Temporal Logic Model Construct.

When created, the TTLM can be saved into the database. Before a rule is saved, the user will be asked to enter the database in which the rule is to be saved, "R" to save

in database **RuleDataBase**. "B" in **BehaviorDataBase**. or "C" in **ControlDataBase**. When the rule is saved, the text file as well as the graphical file are updated to keep them consistent with one another. The text file is the input file to the inference engine when it reasons about the system, and the graphical format file is used when a TTLM is retrieved for reusing, modification, or simply having a look at it.

When the two PSNs are idle, that is **NODE1ID NODE2ID**, another event can occur. This event is: **NODE1** *responds* to an incoming call packet received and according to Figure 6.2 the transition is from the state **NODE1ID NODE2ID** to the state **NODE1DT NODE2ID**.

The logical operator AND is created the same way as well as the remaining statements whose predicate name is RESPONDS, the arguments are NODE1 and CALL, and the temporal operator is UNTIL. The last statement goes attached to the logical operator, which before being saved, has got a conclusion attached.

The TTLMs can be looked at, and corrected by retrieving them. This feature will be explained in mode detail later in this chapter.

Right after all the TTLMs have been saved, the user is able to run the simulation process and create the reachability graph for the uncontrolled system. This is achieved by selecting the first submenu option from the menu in Figure 5.8. The Inference Engine takes the knowledge base of the uncontrolled system and reasons about the system generating a set of uncontrolled states and their corresponding set of events. If requested by the user, this set of states can be plotted onto the screen one state at a time, that is, the first state is created and placed interactively somewhere within the screen space, then the second state is created and similarly placed on the screen, and so on and so forth until the last state is created. At this point, the reachability graph is, then, built (Figure 6.5) and is available both in a graphical form to be displayed on the screen and in a textual form in a file that can be edited using one of the menu selections as explained in Section 5.3.7.

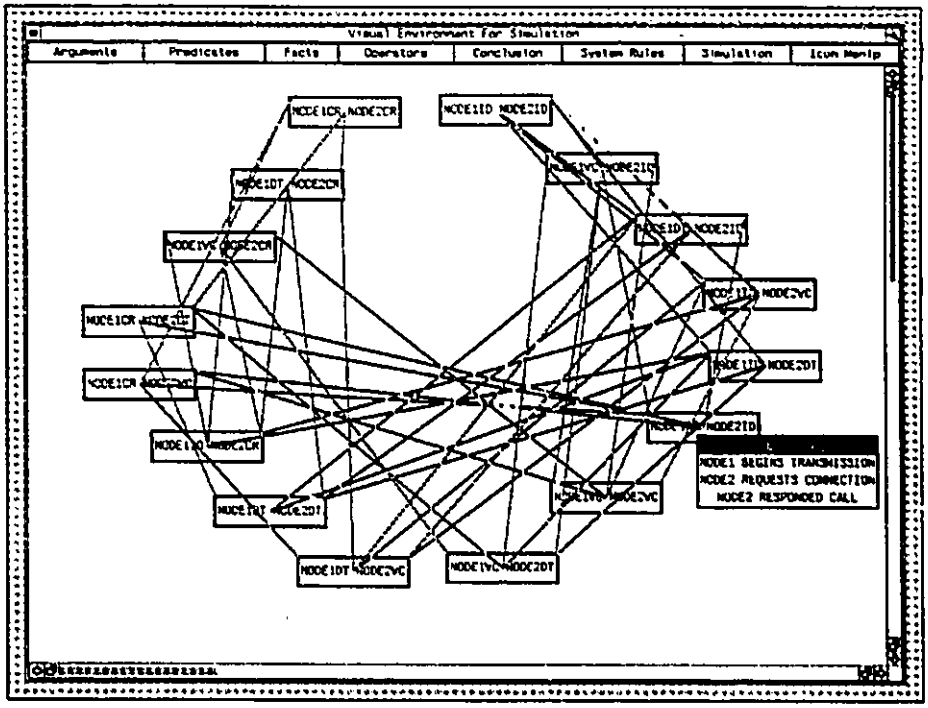

Figure 6.5: Reachability Graph of the Plant or Uncontrolled System.

6.2.2 Controller Specifications

Figure 6.5 shows the reachability graph of the uncontrolled system or Plant. The behavior of the uncontrolled system or DES plant is usually unsatisfactory. A description of the required behavior is imposed to prevent the system from unsatisfactory behavior. This description is also formulated in TTLMs. For instance, formula *6.2.CL1*, shown below, says that when a PSN is at a state in which the connection is established and ready for data transfer (**CR**), the other PSN cannot be at state either **CR** or **WC**.

Let $EQ(x, y)$ represent that x is identical to y. The specifications for the required behavior of system of two entities are as follows:

$$\Box[CR(x) \wedge \neg EQ(x, y) \Rightarrow \neg CR(y) \wedge \neg WC(y)] \qquad (6.2.CL1)$$

$$\Box[DT(x) \wedge \neg EQ(x, y) \Rightarrow \neg WC(y)] \qquad (6.2.CL2)$$

$$\Box[WC(x) \wedge \neg EQ(x,y) \Rightarrow \neg CR(y) \wedge \neg DT(y)] \qquad (6.2.CL3)$$

where $\neg$ is the negation.

The GPE allows the user to inspect the required behavior of the system by selecting the *System Behavior* option from the menu in Figure 5.7,which will allow to retrieve or delete any TTL rules. Some suggestions are given for the controller design. How to look at the suggestions file will be described in the next section.

It is the controller's task to prevent the plant from getting into that unsatisfactory behavior. For this, the set of events at every one of the states is partitioned into disjoint sets of permissible events and unpermissible events, so that the controller will enable or disable those events accordingly. Hence, the controller has the same set of events as that of the plant.

Control theory says "the controller itself is a deterministic plant". In the design of a controller for a nondeterministic DES, the objective is to ensure that the requirements of the desired system behavior are met. Thus, the controller must ensure that the properties such as mutual exclusion, priority, and/or precedence relation are satisfied.

Once the suggestions are clear, the user can start building the controller. The controller consists of set of temporal logic models constructed according to the required behavior of the closed-loop system. They are built in the same way as the TTLMs for the plant were built.

Following the notation in [36], in this thesis we use the local variable symbols p, q, and r to represent the data stored by the controller. q is assigned values 1 and 0 representing if an entity is in state CR or not; p and r are assigned non-negative integer values representing the number of entities which are in state WC or state DT. Let $ADD1(n)$ stand for that n is increased by 1 and $MIN1(n)$ for that n is decreased by 1. Then, the specifications of the controller are as follows:

Dynamics of the Controller

$$\Box[requests(x) \wedge EQ(q,0) \Rightarrow ADD1 \bigcirc (p) \wedge EQ(\bigcirc q,0)] \qquad (6.2.C1)$$

$$\Box[expires(x) \wedge \mathcal{U}(13 \leq \kappa \leq \infty) \wedge \neg EQ(p,0) \Rightarrow MIN1 \bigcirc (p)] \qquad (6.2.C2)$$

$$\Box[confirmed(x) \wedge \mathcal{U}(4 \leq \kappa \leq 7) \wedge EQ(p,1) \wedge EQ(q,0) \Rightarrow EQ(\bigcirc p,0) \wedge EQ(\bigcirc q,1)] \quad (6.2.C3)$$

$$\Box[responds(x) \wedge \mathcal{U}(0 \leq \kappa \leq 13) \wedge EQ(q,1) \wedge EQ(r,0) \Rightarrow EQ(\bigcirc q,1) \wedge EQ(\bigcirc r,1)] \quad (6.2.C4)$$

$$\Box[begins(x) \wedge \mathcal{U}(11 \leq \kappa \leq 13) \wedge EQ(q,1) \wedge EQ(r,1) \Rightarrow EQ(\bigcirc q,0) \wedge EQ(\bigcirc r,2)] \quad (6.2.C5)$$

$$\Box[finishes(x) \wedge \mathcal{U}(11 \leq \kappa \leq 13) \wedge \neg EQ(r,0) \Rightarrow MIN1 \bigcirc (r)] \qquad (6.2.C6)$$

Initial Condition of the Controller

$$EQ(p,0) \wedge EQ(q,0) \wedge EQ(r,0) \qquad (6.2.C7)$$

As already mentioned, the knowledge base of the controlled system is stored in a different database, **ControlDataBase**. To create the set of states of the controlled system, the inference engine is run, but in this case, it reasons over the rules of the knowledge base of the controlled system.

The states of the controlled system can be also displayed; this is achieved in two ways. If the states of the uncontrolled system exist on the screen, see Figure 6.5, then all the existent connections are deleted, and the states of the controlled system will be displayed automatically on top of their corresponding states of the uncontrolled system, and again the connections of the states are established, see in Figure 6.6. As for the uncontrolled system, the textual representation of the reachability graph can be obtained by selecting the corresponding submenu selection from the bar menu.

Comparing Figure 6.5 and Figure 6.6, in Figure 6.6 there are fewer states as the unreachable states have been left out. The number of event at the states would be probably less too because some events are disabled. The set of events can be seen interactively by pressing the right button of the mouse devise on the desired state.

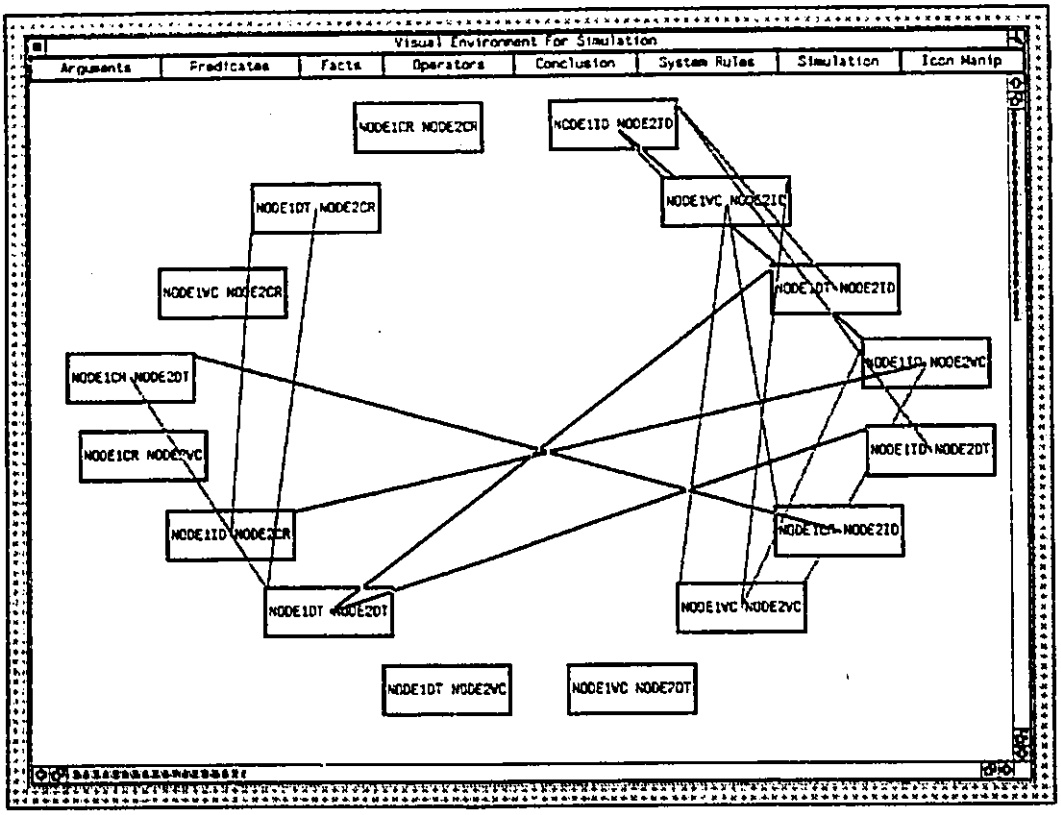

Figure 6.6: Reachability Graph of the Controlled System (Closed-Loop).

In the case that the uncontrolled system has not been placed on the screen, the set of states of the controlled system are displayed interactively on the screen as they are created, as it was mentioned above, since no information about the objects' coordinates is available.

6.2.3 Optimal Path

The optimal path is a sequence of events that drives the system along a trajectory from an initial state to a final state which is reachable from the initial state that has a minimum value of cost. Finding the optimal path is described in the next section.

Lin in [36] defines the cost function by

$$J(s_0, s_g) = \sum_{i=0}^{g-1} \theta(e_{i+1}, s_i) \tag{6.2.1}$$

for all trajectories starting from s_0 and ending at s_g. In the same citation, it was proved

that the A^* algorithm can select a sequence of events to drive the system from the initial state s_0 to the final state s_g with the minimal cost.

To implement the A^* algorithm, the costs of events (distance from the current state to some goal state) and the preference indexes of the system (distance from the current state to the goal state) are needed, they are shown in Table 6.1 and Table 6.2 respectively. And the cost of maintaining the system at any state is assumed to be 0.

event e	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	e_{10}	e_{11}	e_{12}
cost θ	1	2	3	4	1	2	3	4	5	6	1	2

Table 6.1: The cost function.

state S_i	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_9	S_{10}	S_{12}	S_{14}
$index(s_i)$	0	1	2	2	2	3	3	4	4	4	5

Table 6.2: The index function.

Figure 6.7 shows the optimal path from the initial node which is the state **NODE1ID NODE2ID**. From the initial state the search mechanism chooses the best or one of the best alternative directions for searching, according to the evaluation function f. Choosing the best direction is performed until the final or goal state is found, that, in this case, is **NODE1DT NODE2DT**.

6.2.4 Real-Time Simulation

To visualize the simulation outcome is very important when the controller is being designed since those results are taken to accept or modify the design to be used for other simulation experiments.

Consider the reachability graph for the controlled system, shown in Figure 6.6. The menu selection *Automatic Simulation* can be chosen from the list of submenu options shown

in Figure 5.8. If this option is chosen, the user will be prompted to enter the initial state from which the simulation will start. This object chosen belongs to the StateCls class. As mentioned before, each instance of class StateCls has a variable that encapsulates a set of events at that state.

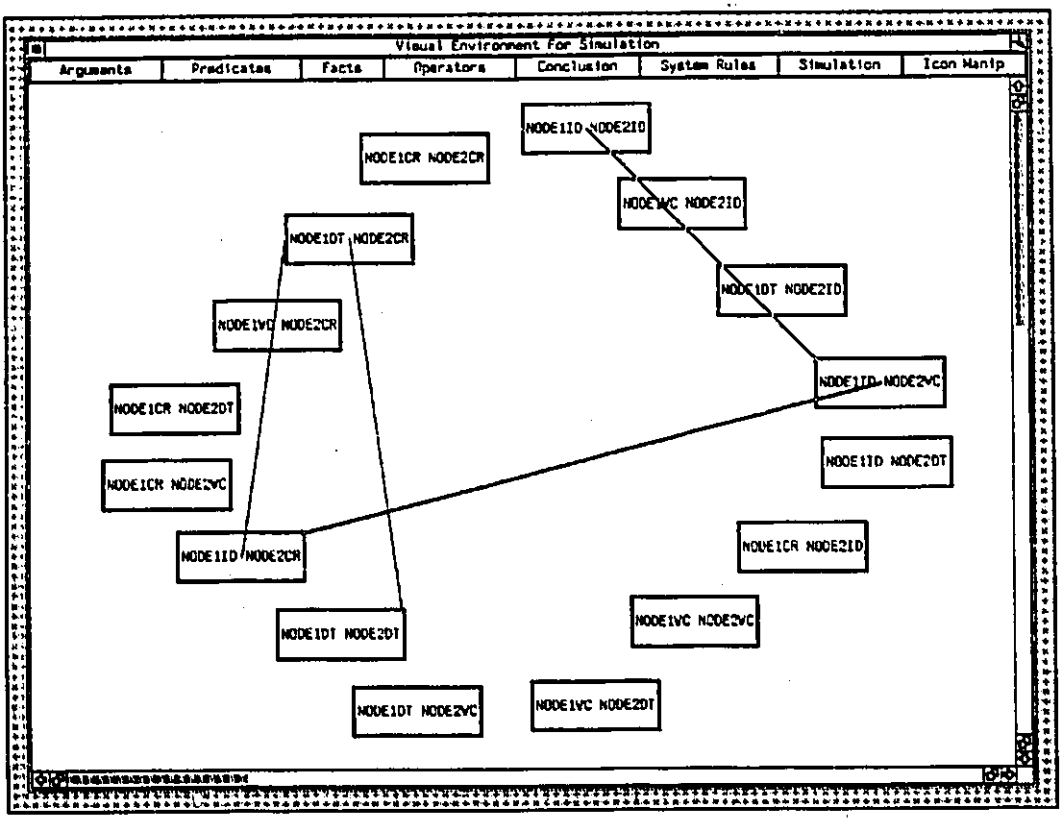

Figure 6.7: Optimal Path of the Controlled System.

A method to randomly generate events is implemented. This method uses a C function that returns non-negative long random integers. An event is chosen from the set of events according to the number returned by the random number generator function. That is, an ordered collection of objects is used to store all the events of the system, and the random number will point to one of this events.

The random event generation method is called from any state of the system to generate an event to keep the system on going according to its behavior. Meanwhile, a timer (clock) is updated or increased as a means of time measurement for real time simulation. Then,

the event just generated is checked to see if it belongs to the set of permissible events of the state at which the system is. Two situations can be found: that the event belongs to the state's set of events or it does not.

If the newly generated event is a permissible one, the lower time bound and the upper time bound, set by the temporal operator, are also checked against the clock. If the clock value is in between the event's time boundaries, the event is displayed right beneath the icon which represents graphically the state, and the process automatically goes to the next state indicated by the event, this state will be highlighted. If the clock value is less than the lower bound, meaning that the next state is unreachable at that discrete time, then the event will wait until the clock reaches a value that at least equals the lower time bound for the state transition to take place. If the clock value is greater than the upper time bound, no state transition will occur and the process will go back to the state from where the system started.

If the generated event does not belong to the state's set of permissible events, another event is generated by the function until a permissible event is generated or a timeout signal has occurred.

Figures 6.8 and 6.9 show part of the sequence of the automatic simulation. Figure 6.8 shows that at state **NODE1WC NODE2ID** the permissible event *NODE1 CONFIRMED CONNECTION* is being generated and analyzed in terms of the clock and its time boundaries. That event triggers the process to go to the next state, that is, the state NODE1CR NODE2ID. Figure 6.9 shows the next state highlighted indicating the state to which the process is. This is repeated until a very high number of events has been generated or the user stops it by clicking any of the button of the mouse device. This animated feature allows the user to check for malfunctions of the DES simulation model.

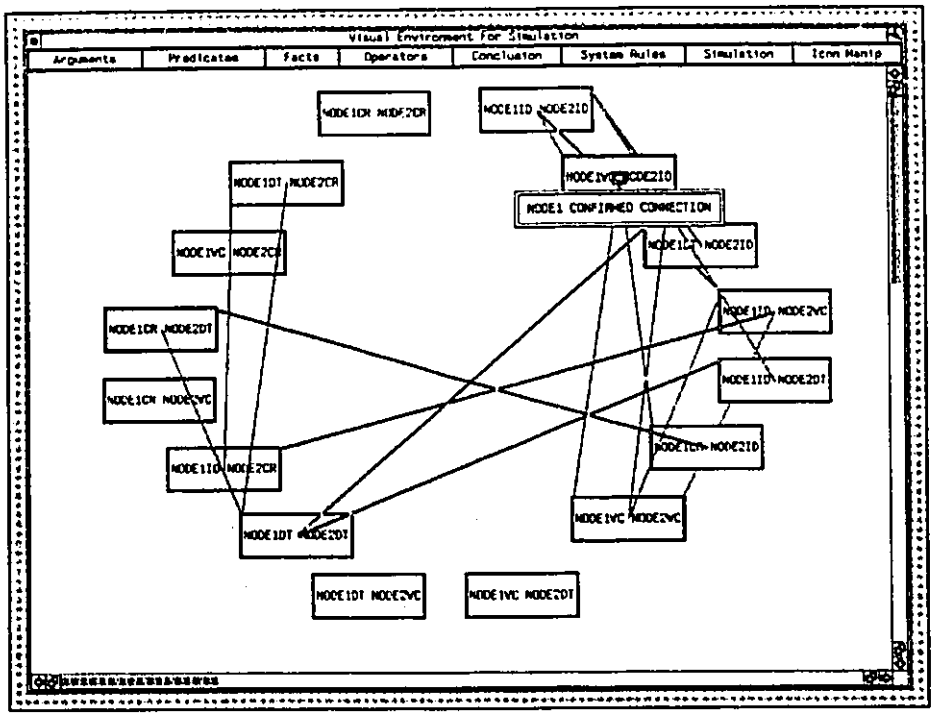

Figure 6.8: Automatic Simulation: Event randomly Generated.

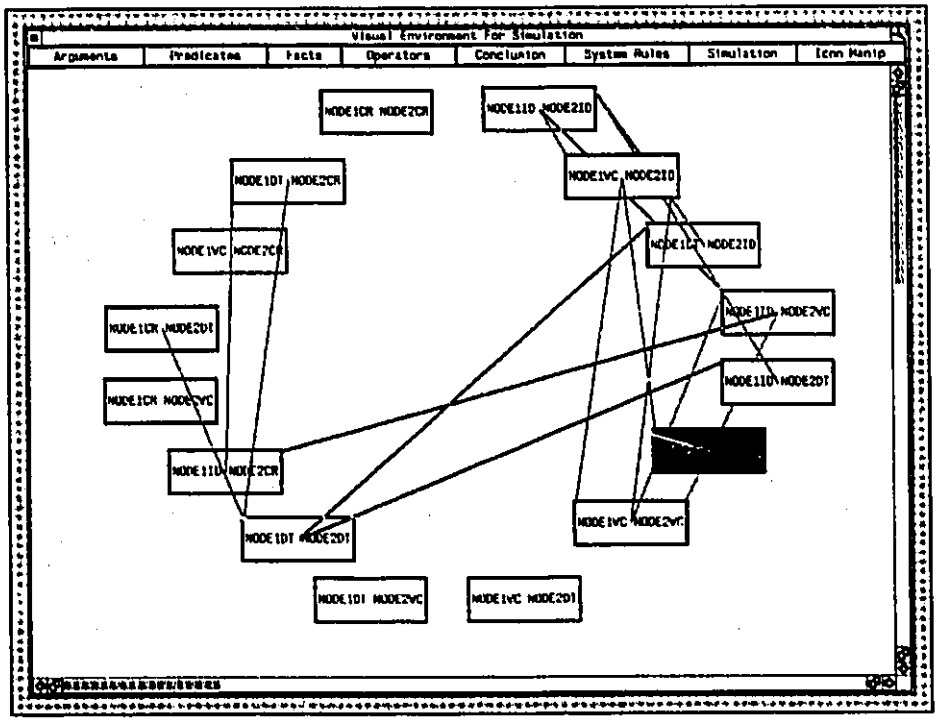

Figure 6.9: Automatic Simulation: Next State.

6.3 Example 2: Telephone System

The specification of a telephone system is another example implemented using this TTL language. A complete specification a Telephone System (TS) is described in [17]. Figure 6.10 shows the specifications (state-transition diagram) of the telephone.

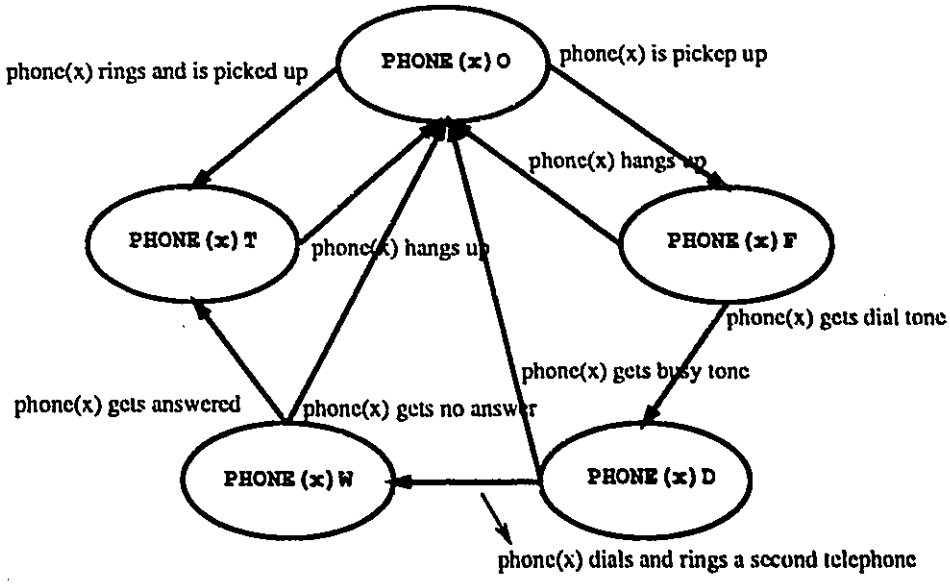

Figure 6.10: State-Transition of the Telephone System.

The local variable x represents the states of the protocol entities. The system has five states. **Phone(x)O** which means that the handset is on the hook; **Phone(x)F** which stands for handset off the hook; **Phone(x)D** means that a number is being dialed; **Phone(x)W** which stands for waiting for an answer; and **Phone(x)T** stands for talking the other party.

The system simulated in this section consists of two telephones that can access an arbitrary number of users, and at the same time they can be accessed by an arbitrary number of users. To establish a talk or connection between two sites, the first user (the caller) picks up the handset. The exchange switching network responds by sending a dial tone to the caller. Now the caller can dial the phone number of the second user (or the called). After completing dialing, the caller waits until the exchange switching network checks the status of that phone number and returns a audible tone. That tone signal can

be of two types: busy tone, if the second user has the handset off hook, and ring signal otherwise. If the second phone is picked up, the ring signal stops and the two sites can begin the conversation.

There are nine events that can occur at different states. The event *ripickedup(x)*, phone(x) rings and is picked up and the event *pickedup*, phone(x) is picked up can only occur at the state when phone(x) is on the hook. At state phone(x) is off hook, the events that can occur are: *getsdialtone(x)*, phone(x) get dial tone and *hungup(x)*, phone(x) is hung up. At state phone(x) is dialing a number, the events that can occur are: *getsbusytone(x)*, phone(x) gets the busy tone and *rings(x)*, phone(x) ringing a second party. At state phone(x) is waiting to be answered, the events *getsanswer(x)*, phone(x) gets an answer from a second party and *getsnoanswer(x)*, when nobody answers the other phone, can only occur. At the last state, phone(x) talks, the only event that can occur is *hungup(x)*, or phone(x) is hung up.

Figure 6.10 shows the diagram transition of a Telephone System. A feature has been added to the specifications of the system described above for two telephones, this feature is *HOLD*. While a party is talking on the phone, an incoming call can be received. This party is signaled by beeps on the handset's speaker that a phone call is coming. The telephone switch must momentarily be depressed in order to answer the incoming call and engage in conversation with the third party, meanwhile the second party is place on hold. The person who placed the hold has to again depress the telephone switch in order to take back the original call. Figure 6.11 shows the state-transition of the system including this feature. The state **PHONE(x)** has been added to represent the fact that a party that was talking has been put on hold by a second party who engaged in conversation with a third one. Also two more events are considered to occur: at state when phone(x) is talking, the event *hold(x)* that means it is put on hold can occur; and at state phone(x) is on hold, the event *backfromhold(x)* can occur.

A Telephone System is formed of number of stations which can perform the two ba-

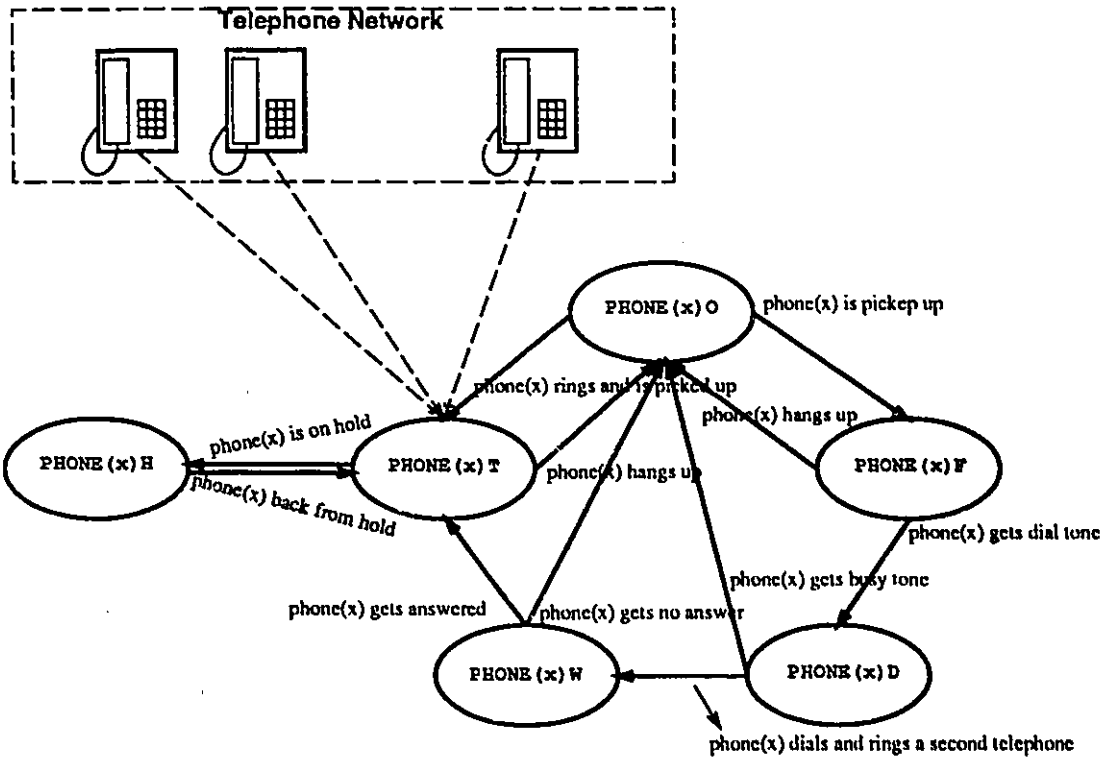

Figure 6.11: State-Transition with New Feature: HOLD.

sic roles: call generation and call reception, and the exchange network that provides the switching interfaces necessary to connect or disconnect two telephone stations. Each station has a number of internal variables for telephone features that are set according to user requests.

A caller generates a call by taking the handset off the hook and waiting to receive the dial tone from the exchange or switching network. Once a station is off the hook it can be hung up at any time. On the dial tone, the caller dial the digits of the destination to request connection which is provided by the exchange. The switching network will check the destination's status, if the destination is idle, it rings the destination signaling an incoming call, if busy, the switching network checks the variable that sets the hold feature that will lead the switching network to signal the destination about the existence of an incoming call, or send the busy tone to the caller, otherwise.

If the feature hold is set, when having an incoming call, the user will hear a beep on

the handset. The connection is established with the incoming call by pressing and releasing the switch hook of the station leaving on a hold state the other party.

Another way to start a conversation is by having an incoming call. In the event of having an incoming call, after the handset is lifted up, the exchange makes the connection with the call initiator.

The feature introduced adds complexity to the plant specification of the Telephone System. This complexity is due basically to the fact that some states that were forbidden are now possible to occur, the addition of a new state, and some other events. For instance, even though someone is talking on the phone, the HOLD feature allows a third party to beep on any handset of the network that is engaged in conversation instead of getting the busy tone.

The specifications are written using the TTL language described in Chapter 2 and used in the previous section to model the packet-switched communication protocol. The specifications in TTLMs are as follows:

Dynamics of the Plant

$$\Box[(PHONE(x)O \wedge \mathit{pickedup}(x)) \wedge \mathcal{U}(0 \leq \kappa \leq 7) \Longrightarrow PHONE \bigcirc (x)F] \qquad (6.3.P1)$$

$$\Box[(PHONE(x)O \wedge \mathit{ripickedup}(x)) \wedge \mathcal{U}(0 \leq \kappa \leq 7) \Longrightarrow PHONE \bigcirc (x)T] \qquad (6.3.P2)$$

$$\Box[(PHONE(x)F \wedge \mathit{getsdialtone}(x)) \wedge \mathcal{U}(7 \leq \kappa \leq 12) \Longrightarrow PHONE \bigcirc (x)D] \qquad (6.3.P3)$$

$$\Box[(PHONE(x)F \wedge \mathit{hungup}(x)) \wedge \mathcal{U}(7 \leq \kappa \leq 10) \Longrightarrow PHONE \bigcirc (x)O] \qquad (6.3.P4)$$

$$\Box[(PHONE(x)D \wedge \mathit{getsbusytone}(x)) \wedge \mathcal{U}(12 \leq \kappa \leq 17) \Longrightarrow PHONE \bigcirc (x)O] \qquad (6.3.P5)$$

$$\Box[(PHONE(x)D \wedge \mathit{rings}(x)) \wedge \mathcal{U}(17 \leq \kappa \leq 20) \Longrightarrow PHONE \bigcirc (x)W] \qquad (6.3.P6)$$

$$\Box[(PHONE(x)W \wedge \mathit{getsanswer}(x)) \wedge \mathcal{U}(20 \leq \kappa \leq 25) \Longrightarrow PHONE \bigcirc (x)T] \qquad (6.3.P7)$$

$$\Box[(PHONE(x)W \wedge \mathit{getsnoanswer}(x)) \wedge \mathcal{U}(20 \leq \kappa \leq 30) \Longrightarrow PHONE \bigcirc (x)O] \qquad (6.3.P8)$$

$$\Box[(PHONE(x)T \wedge \mathit{hungup}(x)) \wedge \mathcal{U}(25 \leq \kappa \leq 28) \Longrightarrow PHONE \bigcirc (x)O] \qquad (6.3.P9)$$

$$\Box[(PHONE(x)T \wedge hold(x)) \wedge \mathcal{U}(28 \leq \kappa \leq 32) \Longrightarrow PHONE \bigcirc (x)H] \quad (6.3.P10)$$

$$\Box[(PHONE(x)H \wedge hungup(x)) \wedge \mathcal{U}(32 \leq \kappa \leq 35) \Longrightarrow PHONE \bigcirc (x)O] \quad (6.3.P11)$$

$$\Box[(PHONE(x)H \wedge backfromhold(x)) \wedge \mathcal{U}(32 \leq \kappa \leq 37) \Longrightarrow PHONE \bigcirc (x)T] \quad (6.3.P12)$$

Let $EQ(x, y)$ represent that x is identical to y. The required behavior of the system is as follows:

$$\Box[PHONE(x)W \Rightarrow \neg EQ(x, y) \wedge \neg PHONE(y)W \wedge \neg PHONE(y)F] \quad (6.3.CL1)$$

$$\Box[PHONE(x)T \Rightarrow \neg EQ(x, y) \wedge \neg PHONE(y)F] \quad (6.3.CL2)$$

$$\Box[PHONE(x)F \Rightarrow \neg EQ(x, y) \wedge \neg PHONE(y)W \wedge \neg PHONE(y)T] \quad (6.3.CL3)$$

$$\Box[PHONE(x)F \Rightarrow \neg EQ(x, y) \wedge \neg PHONE(y)H] \quad (6.3.CL4)$$

As in the previous section, we use the local variables p, q, r, and t to represent the data stored by the controller. q is assigned values of 1 and 0 representing if a telephone is at state PHONE(x)W or not; p, r and t are assigned non-negative integer values representing the number of telephones which are at state PHONE(x)F, or state PHONE(x)T, or state PHONE(x)H. Let $ADD1(n)$ stand for that n is increased by 1 and $MIN1(n)$ for that n is decreased by 1. Then, the specifications of the controller are as follows:

Dynamics of the Controller

$$\Box[pickedup(x) \wedge EQ(q, 0) \Rightarrow ADD1 \bigcirc (p) \wedge EQ(\bigcirc q, 0)] \quad (6.3.C1)$$

$$\Box[getsnoanswer(x) \wedge \mathcal{U}(0 \leq \kappa \leq 22) \wedge \neg EQ(p, 0) \Rightarrow MIN1 \bigcirc (p)] \quad (6.2.C2)$$

$$\Box[rings(x) \wedge \mathcal{U}(0 \leq \kappa \leq 22) \wedge EQ(p, 1) \wedge EQ(q, 0) \Rightarrow EQ(\bigcirc p, 0) \wedge EQ(\bigcirc q, 1)] \quad (6.3.C3)$$

$$\Box[rings(x) \wedge \mathcal{U}(0 \leq \kappa \leq 22) \wedge EQ(q, 1) \wedge EQ(r, 0) \Rightarrow EQ(\bigcirc q, 1) \wedge EQ(\bigcirc r, 1)] \quad (6.3.C4)$$

$$\Box[getsanswer(x) \wedge \mathcal{U}(0 \leq \kappa \leq 22) \wedge EQ(q, 1) \wedge EQ(r, 1) \Rightarrow EQ(\bigcirc q, 0) \wedge EQ(\bigcirc r, 2)] \quad (6.3.C5)$$

$$\Box[hangup(x) \wedge \mathcal{U}(0 \leq \kappa \leq 22) \wedge \neg EQ(r,0) \Rightarrow MIN1(\bigcirc r)] \qquad (6.3.C6)$$

$$\Box[hangup(x) \wedge \mathcal{U}(0 \leq \kappa \leq 22) \wedge \neg EQ(t,0) \Rightarrow MIN1 \bigcirc (t)] \qquad (6.3.C7)$$

Initial Conditions of the Controller

$$EQ(p,0) \wedge EQ(q,0) \wedge EQ(r,0) \wedge \mathcal{U}(0 \leq \kappa \leq 22) \qquad (6.3.C8)$$

Next, a complimentary number of features introduced in the GPE are described. Some of these features are: to retrieve or delete a rule from any of the databases, to compute and display the reachability graph of the uncontrolled and controlled system, to find and display the optimal path.

6.3.1 Accessing the System Rules

In Chapter 5 Figure 5.13 shows the bar menu option *System Rules* with its three menu selections: The general system (plant), the system behavior, and the controller.

The first menu selection is regarding the rules of the plant in which three submenu options are found.When the pointer devise is placed inside the option *Retrieve Rule* a popup menu will show up. This popup menu contains the rules of the plant that have been saved.

Figure 6.12 shows the popup menu from which rule #13 has been chosen and displayed on the screen. Rules can be retrieved for three reasons: to correct mistakes, to change information, or simply curiosity to have a look at it.

The second menu selection *System Behavior* is the menu interface to the information about the system behavior of the system. Rules can be retrieved, deleted, or a file with the textual format of the rules can be edited.

Some suggestions that help in designing the controller are given by the monitor. The monitor bases its suggestions on the information stored in the database of the system behavior. These suggestions advise the user about unreachable states as well as unpermissible events. A feature to edit the file with this information in discussed later in this chapter.

To remove a rule from the database of the system behavior, the mouse devise should be dragged until the *Remove Rule* menu option. Once the mouse is placed on this menu option, a popup menu will appear. This menu is similar to the one that pops up when a rule is being retrieved. This is the menu from which the rule to be deleted is chosen.

From the last menu *Controller Behavior* the rules that shape the controller can be retrieved, deleted, or a file edited whose content is the textual form of the rules written in the TTL language.

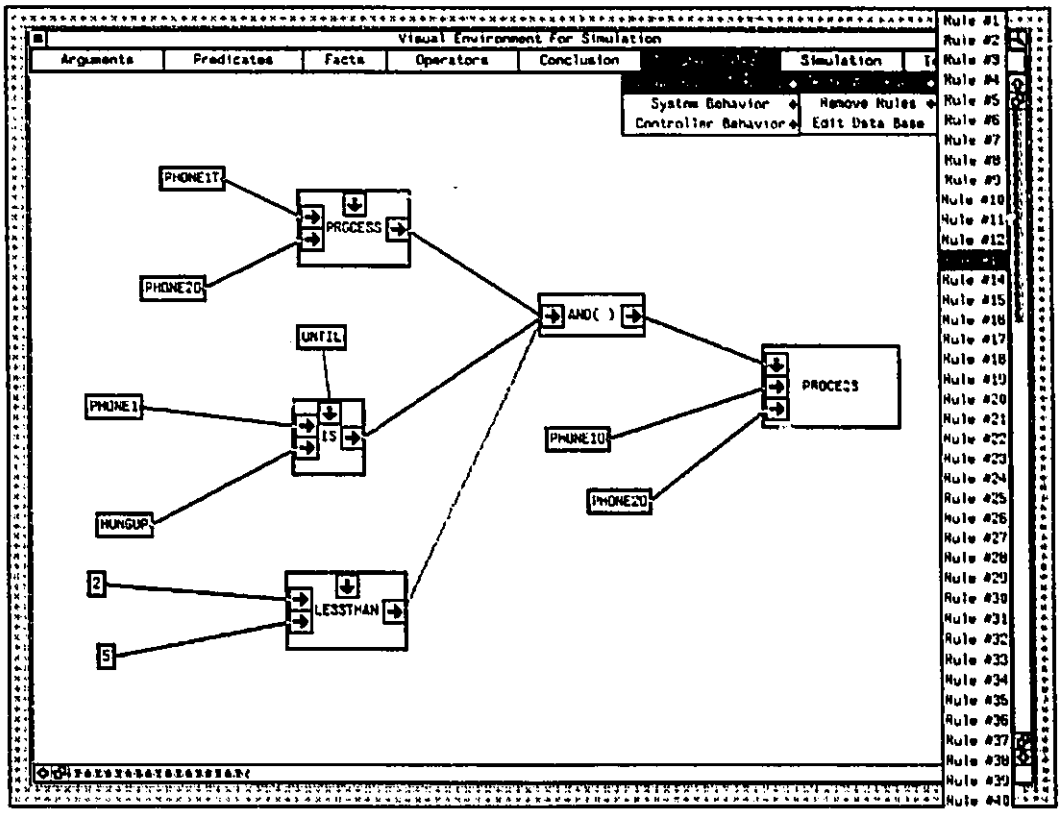

Figure 6.12: A Rule Being Retrieved.

The uncontrolled system and the controller should work concurrently and that action is achieved by synchronizing the events to occur. When the controller is synchronized with the uncontrolled behavior of the system, certain selected events that will take the system to undesired behaviors are prevented from occurring. Those events that take the system to undesired behaviors are detected by the monitor and written in the suggestion list, this

feature will be explained in the Sub-section 6.3.3. Figure 6.13 shows the file that contains the specifications of the controller in the textual format.

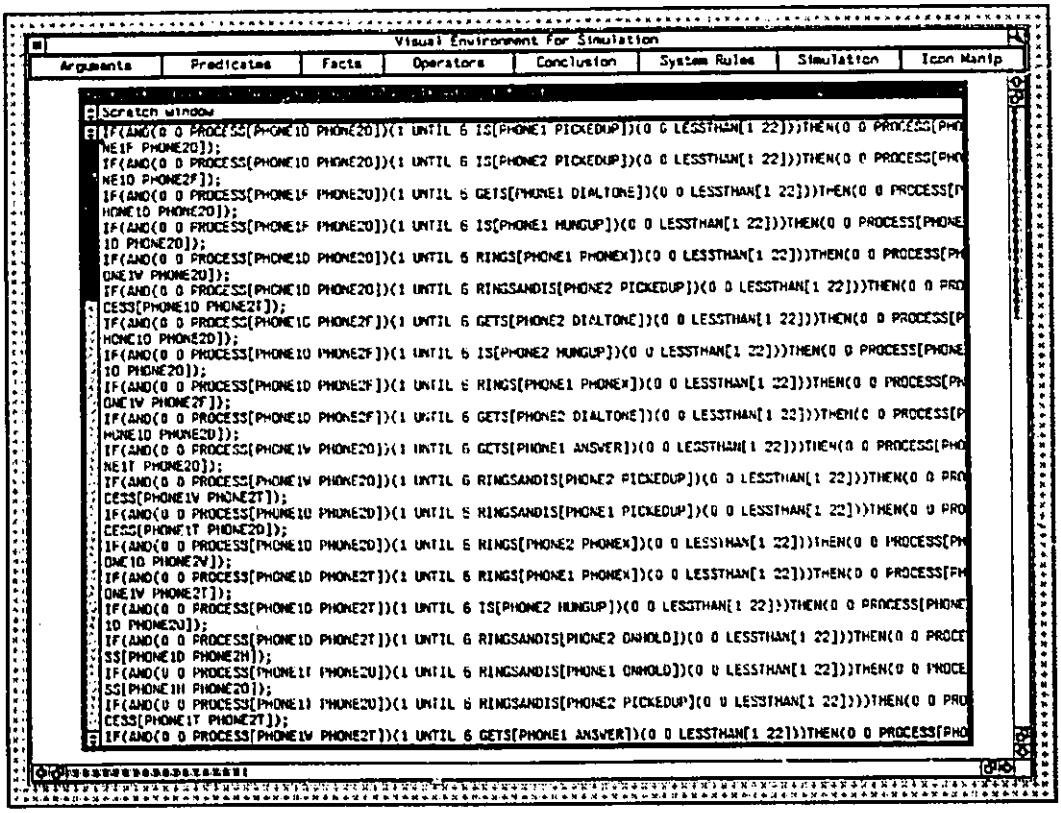

Figure 6.13: Textual Format of the Controller Behavior.

6.3.2 Simulation

The menu selection *Simulation*, shown in Figure 5.14, adds some other features to the GPE. These features allows the user to run simulations for DESs, display the resultant reachability graphs of those simulations, find and display the optimal path in the reachability graph.

6.3.2.1 Simulating

The option *Simulating* allows the user to run the DES simulation process. When this option is chosen, the main routine is called. The main routine contains in a sequence all the code lines to run the simulation, this sequence is as follows:

1. Create the database of facts or initial events.
2. Create the database of rules.
3. Perform reasoning on the database of rules for the plant,
4. Create the states of the plant and store them in an ordered collection.
5. Perform reasoning on the database of rules for the controller behavior.
6. Create the states of the controlled system.

6.3.2.2 Uncontrolled and Controlled System

The states of both the uncontrolled system (plant) and the controlled system are created. The reachability graph of both systems (uncontrolled and controlled) can be displayed when the the *Create States* option is chosen from the menu option *Uncontrolled System* and *Controlled System*. In the previous section, the reachability graphs for both the uncontrolled and the controlled system of the packet-switched network were displayed.

6.3.2.3 Editing Suggestions File

A mechanism to monitor the process is implemented. This mechanism gives some suggestions or warnings about those states that should not be reached since they might lead to deadlocks and those events that should be disabled. Based on these suggestion the controller is designed.

Figure 6.14 shows the file containing the suggestions to build the controller. In the figure, it can be seen all the states that should be unreachable as for example the states 2, 4, 7, 9, 18, 23, 27, 34, 35 of the reachability graph. The set of events that should be disabled and the corresponding state, for instance, the events phone1 rings and is picked up and phone2 rings and is picked up should be disabled at state 0.

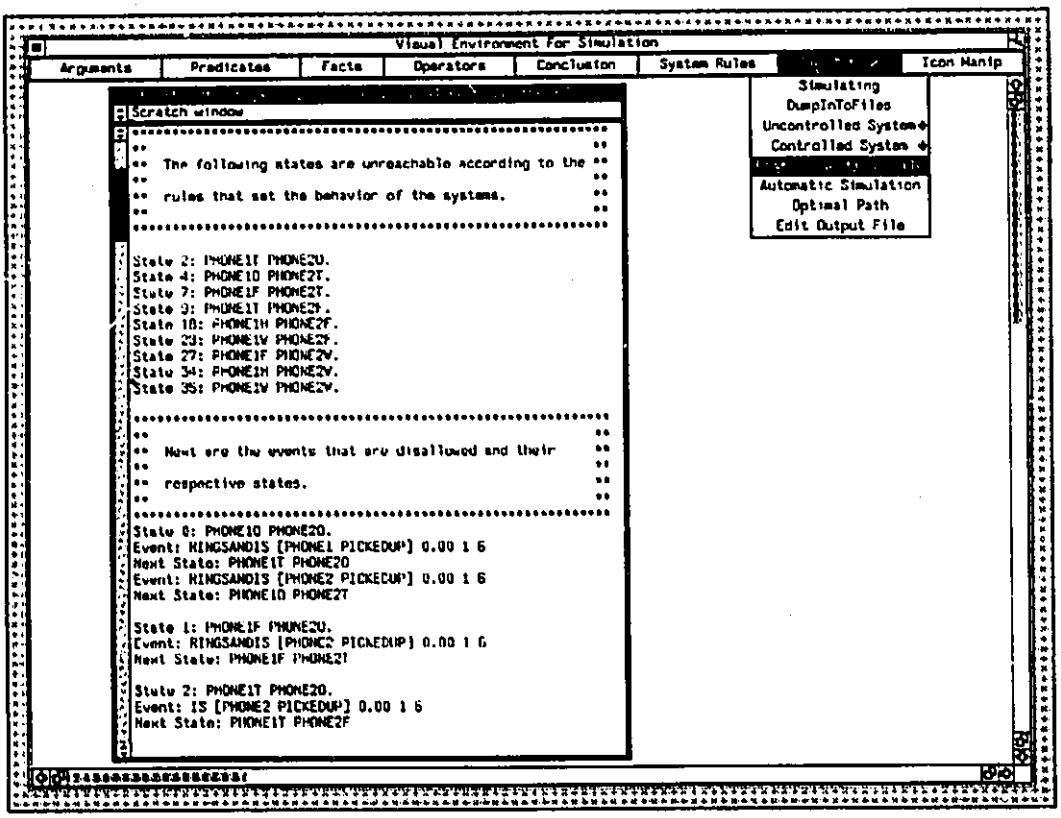

Figure 6.14: Warnings and Suggestions Given by the Monitor.

6.3.2.4 Automatic Simulation

Another feature is *Automatic Simulation*. Timing characteristics have been added to the temporal operator. A global variable (clock) allows to check those timing behaviors of the rules, so that events take place, wait or are rejected. The fact that the caller cannot wait longer than twenty seconds to dial is represented in the TTLM as follows:

$$\Box[(PHONE(x)F \wedge getsdialtone(x))\mathcal{U}(0 \leq \kappa \leq 22) \Longrightarrow PHONE(\bigcirc x)D],$$

and is interpreted as: if the event *getsdialtone* occurs when the process is at the state $PHONE(x)F$, and the clock value is greater than or equal to 0 and less than or equal to 22 the rule is said to hold.

6.3.2.5 Optimal Path

An instance variable has been added to the class Events to store the cost so that every event is given weight value or how expensive is the occurrence of the events. This cost variable allows to generate a sequence of events to drive the process from an initial state to a final state of the controlled system along the desirable trajectory. The reachability graph of the controlled Telephone System is shown in Figure6.15.

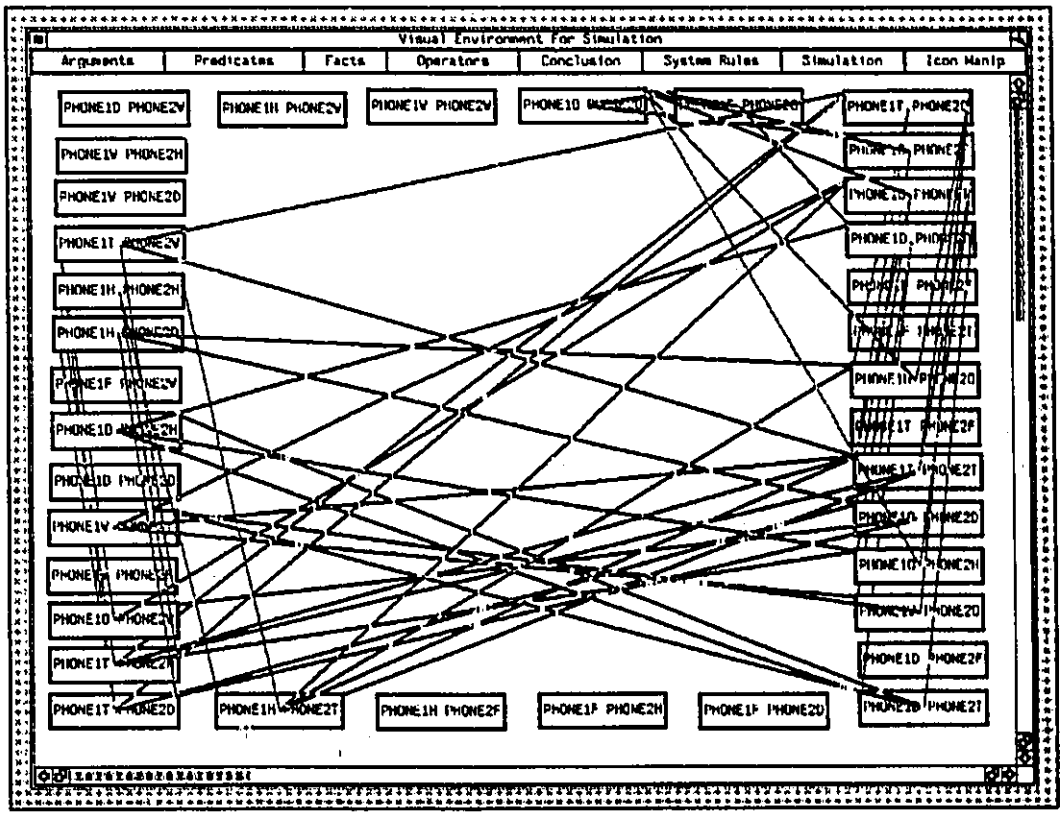

Figure 6.15: Controlled Telephone System.

The purpose of optimization is to produce a guide to accomplish some tasks satisfying the prescribed cost index. The problem of finding the optimal path is solved by a using the

theory of heuristic search by applying the A^* algorithm, described in [60], which with the aid of an evaluation function f on the nodes of the reachability graph and some heuristic indexes finds the optimal path to get to the goal node from the initial node.

Table 6.3 and Table 6.4 show the costs of events and the preference indexes respectively. And the cost of maintaining the system at any state is assumed to be 0.

event e	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	e_{10}	e_{11}	e_{12}
cost θ	1	2	3	4	1	2	3	4	5	6	1	2

Table 6.3: The cost function.

state S_i	S_0	S_1	S_2	S_3	S_4	S_5	S_8	S_{10}	S_{11}	S_{12}	S_{13}	S_{15}
$index(s_i)$	0	1	2	2	2	3	3	4	4	4	5	6

Table 6.4: The index function.

state S_i	S_{19}	S_{20}	S_{21}	S_{22}	S_{24}	S_{26}	S_{28}	S_{29}	S_{30}
$index(s_i)$	5	5	4	4	4	3	2	2	2

Table 6.5: The index function (Cont.)

Figure 6.6 shows the closed-loop system that has the required behavior, but it has not been yet optimized. The optimization is performed by selecting from the submenu list shown in Figure 5.8 the option *Optimal Path*, then the user will be asked to click on the initial state and the goal or final state. Figure 6.16 shows the optimal path for the TS example.

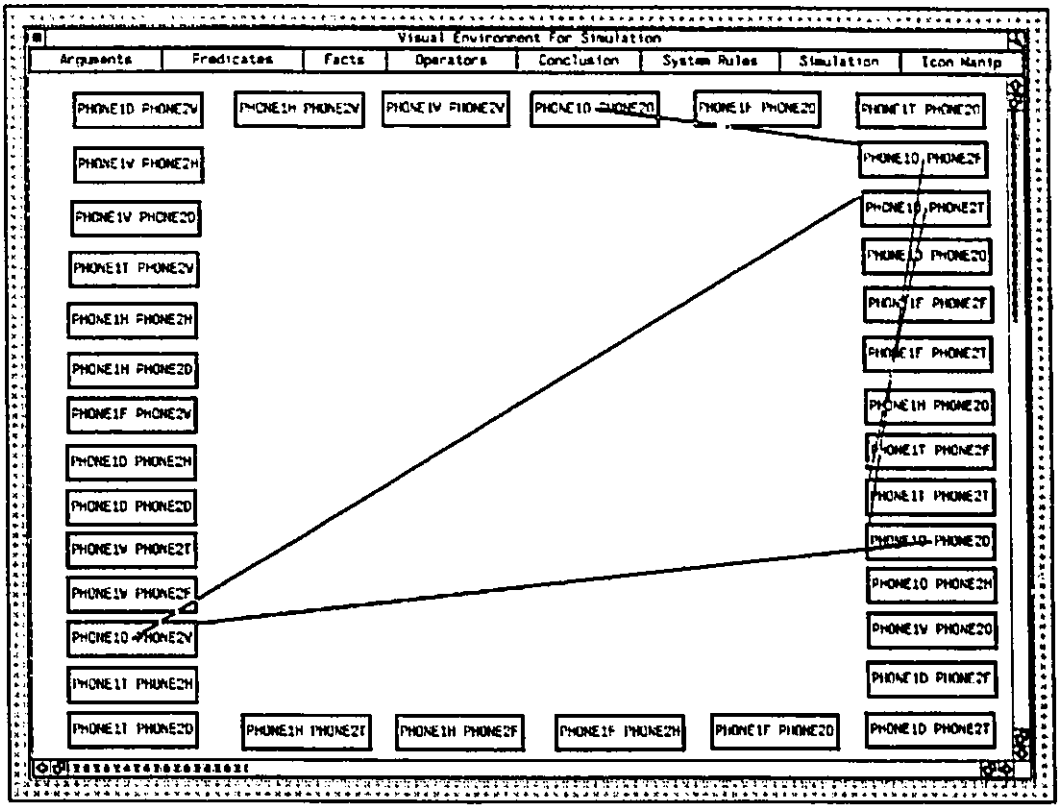

Figure 6.16: Optimal Path of the Telephone System.

6.3.3 Conclusions and Comparison

Two examples have been given to show the advantages of first, using a visual environment to write the temporal logic formulas for systems specification, and second to present the simulation results in a animated fashion.

The fact that the results are presented in a graphical way allows the user to early verify the specification of the uncontrolled system without having the controller designed. If a required behavior is specified within the natural behavior, then a controller or supervisor is needed to act along with the plant in a closed-loop fashion.

A monitor is included to track the behavior of the systems that are naturally described by a trace of certain events. According to some rules, the monitor will draw some suggestions to help in designing the controller. A controller has been designed using the set of

suggestions given by the monitor. This controller will ensure that the system does not get into undesirable states.

The A^* algorithm, provided that the costs of events and the preference indexes of the system are given, is used to will draw the optimal path of the system to go from the initial state to the final state.

The automatic simulation permits to run the simulation in real-time, measuring the time of the system and enabling and disabling events according to value of time. This automatic tool that allows for dynamic verification of the specifications is user-interactive, that is, it can be interrupted by the user at any time if changes are needed and resume running again. When verifying the system specification, the user can find errors more easily by means of the graphical representation of the system simulation.

A work we can make a comparison with is the work presented by Rubin et al. in [56] in which they have used the Visual Programmer's WorkBench (VPW) for a Petri Net/FSA language, called PetriFSA language. With the VPW, they were able to edit, view, execute, simulate, monitor, and animate the PetriFSA diagram for a simple telephone switch example. With the tool designed in this thesis not only can the above functions be performed but also a controller can be designed and synchronized with the running system assuring proper system behavior as well as it interactively finds and shows the optimal path according to a cost function. Since PetriFSA lacks of a controller, the effort to write the specifications is greater because there are no suggestions to consider. On the other hand, our approach counts with a monitor that gives a set of suggestions of all the states that should not be reached and the events that should be disallowed at some states to build up the controller.

Another work we can make a comparison with is the one cited in [17]. They use LOTOS, Language of Temporal Ordering Specifications. In that paper, they write the specifications of a TS using LOTOS. Time aspects something that cannot be dealt with in LOTOS, for instance, the example of having a phone disconnected after the handset has

been off hook for a long time period, let us say 20 seconds, cannot be performed.

Chapter 7

Conclusions and Future Directions

Many concepts such as temporal logic, temporal operators, DESs, real-time processes, and visual programming have been addressed throughout this thesis. This chapter presents a brief summary of the thesis contributions as well as some directions for further research.

7.1 Conclusions

A graphical programming environment (GPE) has been designed to perform simulations of real-time DESs in a timed temporal logic framework. The contributions of this thesis are as follows:

- The temporal logic language defined in [36] has been extended to a Timed Temporal Logic (TTL) language for controlling real-time DESs.
- A clock variable was defined. This clock increases at a constant pace as a formal representation of advancements of time. Using this type of timing measurement (the clock), events and states could be synchronized according to their timing variable such as time bounds and temporal operators. It allows the system to say at any point in time of the simulation whether an event of the set of permissible events of a state can occur or if a state can or cannot be reached.

- The timed temporal logic language is used for the analysis, synthesis, simulation, and optimization of discrete events and real-time systems.

- We have used the concept of object-oriented programming throughout the thesis. An object-oriented framework is used to drive simulations of DESs which among other things involves states, current states, events, current events, next event to be processed, and advance the time. The major advantage of using this programming technique is the fact that all the data pertaining to a state of the DES can be encapsulated in a defined class.

- The GPE assists the user in graphically writing the TTLMs by means of interconnecting icons that represent all the lexical components in which the language has been divided. Those graphical structures are then translated into a textual format that will eventually be parsed for grammar correctness.

- The temporal logic models already parsed can be interpreted by the underlying programming language. This textual format is interpreted by the inference mechanism that executes the reasoning on the system. Thus, the user does not have to know the underlying Objective-C language in order to use the GPE.

- The GPE assists the user in modelling the general system behavior, the required behavior of the system, and the controlled behavior by means of interconnecting graphical symbols that represent the lexical components of the visual language in the TTL framework.

- The GPE allows the user to retrieve the graphical representations of rules and facts, therefore, it increases the user's productivity and allows for a more efficient understanding of the application. The rules and facts can be retrieved to makes changes or corrections on them.

- The reachability set for both the uncontrolled system and the controlled system can be displayed on the screen. The states of which the system consists are created and

plotted on the screen along with the connections to next states corresponding to the event set.

- The optimal path from the initial state to the final state is also viewed using the GPE. It shows the trajectory which the sequence of events generates at minimal cost.
- Simulations in real-time are performed. To achieve such simulations, a global clock variable is used. The timing constraints set by the system specifications are checked and the truth value generated will also depend on the clock value.

7.2 Future Directions

We have been trying to design this Visual Programming Environment as efficient as possible, although it was not of our main concern. However, since the object-oriented paradigm allows to create classes, a lot of improvement in object class definitions can still be incorporated into this system to increase execution speed significantly as well as decrease the number of program code lines.

At the present stage of this research, it is possible to run simulations of systems with a small number of events. But for problems that require more complex interactions among states, other high-level representations of the system are needed as modelling tools to represent the complex situations easily and explicitly. For example, to be able to comprise or encapsulate a number of states into, say, an independent node, and have that node interconnected to some other nodes or single states.

A future research direction for improvements of this visual programming environment is to make it be able to read source and generate the graphical representation of it. This is better known as reverse engineering. By doing this, any modification made in the raw code is directly and immediately reflected in the iconic representation, so that consistency is directly preserved. In the actual implementation, when a change is made in one of the files either the textual or the graphical file of the databases, the two files need to be updated

in order to keep consistency. Above all, this improvement will be beneficial to the user's efficiency by eliminating the need to create and store the graphical representation of the TTLMs.

Since modelling DESs using TTLMs requires more than one database, at least the database of facts and database of rules. To save temporal logic expressions in these databases the interface uses the same method. One suitable solution to this problem would be to be able to create different graphical interface environments and assign each environment to a function.

Bibliography

[1] Abadi M. and Manna Z. "Temporal logic programming". *J. Simbolic Computation*, vol. 8 pp.277-295, 1989.

[2] Abadi M. "The power of temporal proofs" *Theoretical Coputer Science*, vol. 65, 1989, pp 35-83.

[3] Allen J. "Towards a general theory of action and time". *Artificial Intelligence*, vol. 23 no. 2 pp. 123-154, 1984.

[4] Alur R. and Dill D.L. "Automata for modeling real-time systems" *Lecture Notes in Computer Science* vol 443, Springer-Verlag 1990, pp 322-335.

[5] Alur R. and Dill D.L., "The theory of timed automata" *Lecture Notes in Computer Science* vol 600, Springer-Verlag 1990, pp 45-73.

[6] Bacchus F., Tenenberg J, Koomen J.A., "A non-reified temporal logic" *Artificial Intelligence* vol 52, 1991, 87-108.

[7] Baeten J.C.M. and Bergstra J.A., "Real time process algebra" *Formal aspects of Computing* aol 3, no2, 1991, pp 142-188.

[8] Barnes V.M. and Mack J.M. "An algebraic introduction to mathematical logical". Springer-Verlag, New York-Berlin, 1975.

[9] Birtwistle G., Dahl O., Myhrhaoug B., and Nygaard K. "Simula BEGIN". 2nd ed., Studentlitteratur, Lund, 1979.

[10] Bruce B.C. "A model for temporal references and its application in a question-answering program", *Artificial Intelligence*, vol.3, pp.1-25, 1972.

[11] Chang C. and Chang Y. "A real-time distributed simulation of PBX with software reuse". In *Simulation*, vol 54, no. 2, pp 71-79, February 1990.

[12] Clarke E.M., Emerson E.A. and Sistla A.P. "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Programming Languages and Systems*, 8, pp.244-263, 1986.

[13] Dean T. "Using temporal hierarchies to efficiently maintain large temporal Databases". *Journal of the ACM*, 36(4):687-718, 1989.

[14] Dean T.L. and McDermott D. "Temporal data base management", *Artificial Intelligence*, vol.32, pp.1-55, 1987.

[15] Emerson E.A. "Real-time and the mu-calculs" *Lecture Notes in Computer Science* vol 600, Springer-Verlag 1990, pp 175-194.

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

[17] Faci M., Logrippo L., Stepien B. "Formal specifications of telephone systems in LOTOS: the constraint-oriented style approach". *Computer Networks and ISDN Systems*, vol. 21, pp. 53-67, 1991.

[18] Frank R. and Hewitt J. "An object-oriented design approach incorporating simulation in the development of a real-time control system". In *Simulation and the User Interface*, edited by M. A. Life, C. S. Taylor & Francis, 1990.

[19] Fujita M., Kono S., Tanaka T. and Moto-oka T. "Tokio: Logic programming language based on temporal logic and its compilation to prolog". In *Proceedings of the 3rd International Conference on Logic Programming*, pp. 695-709. Springer-Verlag, LNCS 225, 1986.

[20] Fusaoka A., Seki H. and Takahashi K. "A description and reasoning of plant controllers in temporal logic," *Proc. 8th Int. Joint Conf. on Artificial Intelligence*, pp.405-408, 1983.

[21] Gabby D. "Modal and temporal logic programming". In A. Galton, editor, *Temporal Logic and Their Applications*, chapter 6, pp. 197-237. Academic Press, London SW7 2BZ, December 1987.

[22] Gabby D. "The declarative past and imperative future, executable temporal logic for interactive systems". Technical Report, Imperial College, Department of Computing, London SW7 2BZ, January 1989.

[23] Galton A. *Temporal Logics and Their Applications*, Academic Press, London, 1987.

[24] Halsall F. *Data Communications, Computer Networks and OSI*, Addison-Wesley, New York, 1988.

[25] Henzinger T.A., Manna Z. and Pnueli A. "Timed transition systems" *Lecture Notes in Computer Science* vol 600, Springer-Verlag 1990, pp 226-251.

[26] Hirakawa M., Tanaka M., and Ichikawa T. "An iconic programming system, HI-VISUAL".*IEEE Transactions on Software Engineering*, vol.16, no. 10, pp.1178-1184, October, 1990.

[27] Ho Y., Cao X., and Cassandras C. "Infinitesimal and finite perturbation analysis for queueing networks",*Automatica*, vol.19, pp.439-445, 1983.

[28] Jackson P., H. Reichgelt H. and van Harmelen F. *Logic-Based Knowledge Representation*, The MIT Press, Massachusetts, 1989.

[29] Hrycej T. "Temporal prolog". In *ECAI 88 Proceedings of the 8th European Conference on Artificial Intelligence*, Munich, August 1988.

[30] Kahn K. and Gorry G.A. "Mechanizing temporal knowledge", *Artificial Intelligence*, vol.9, pp.87-108, 1977.

[31] Kligerman E. and Stoyenko A. D. "Real-time euclid: a language for reliable real-time languages". *IEEE Transactions on Software Engineering*, vol. 12, no. 9, pp. 941-949, September 1986.

[32] Koymans R. "Specifying real-time properties with metric temporal logic," *Real-Time Systems*, vol.2, pp.255-299, 1990.

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

[34] Lamport L. "What good is temporal logic". In *Proceedings of IFIP, Infromation Processing*, R. E. A. Mason, ed. Amsterdam, The Netherlands: Noth-Holland. 1983. pp. 657-668.

[35] Lamport L. "Specifying concurrent programs modules". *ACM Transactions on Programming Languages and Systems*, vol. 5, pp. 190-222, April 1983.

[36] Lin J. "A temporal logic approach to the analysis and synthesis of discrete event systems". *Doctorade Thesis Submitted to the Electrical Engineering Department at the University of Ottawa*, 1993.

[37] Manna Z. and Wolper P. "Synthesis of communicating processes from temporal logic specifications". *Report No. STAN-CS-81-872*, Standford University, 1981.

[38] Manna Z. and Pnueli A. "Verification of concurrent programs: a temporal proof system". *Report No. STAN-CS-83-967*, Standford University, 1983.

[39] Manna Z. and Pnueli A. "How to cook a temporal proof system for your pet language," *Proc. 10th Annual ACM Symp. on Principles of Programming Languages*, pp.141-154, 1983.

[40] McDermott D. "A temporal logic for reasoning about processes and plans". *Cognitive Science*, vol.6, pp.101-155, 1982.

[41] Manna Z and Wolper P. "Synthesis of communicating processes from temporal logic specifications," *ACM Trans. Programming Languages & Systems*, vol.6, no1, pp.68-93, 1984.

[42] Moszkowsky B. Executing temporal logic programs. Cambridge University Press. Cambridge, 1986.

[43] Orgun M. and Wadge W. "Towards a unified theory of intensional logic programming". Technical Report DCS-112-IR, Department of Computer Science, Univol. of Victoria, B.C., Canada, March 1989.

[44] Ostroff J.S. *Real-Time Computer Control of Discrete Event Systems Modelled by Extended State Machines: A Temporal Logic Approach*, Ph D. thesis, Dept. Electrical Engineering, University of Toronto, 1987.

[45] Ostroff J. "Temporal logic for real-time systems". John Wiley and Sons, 1989.

[46] Ostroff J. "A logic for real-time discrete event processes". *IEEE Control Systems Magazine*, June, pp. 95-102, 1990.

[47] Ostroff J.S. "Deciding properties of timed transition models," *IEEE Trans. Parallel and Distributed Systems*, vol.1, pp.170-183, 1990.

[48] Ozden M. "Graphical programming of simulation models in an object-oriented environment". *Simulation*, February, pp. 104-116, 1991.

[49] Pasztor A. and Sain I. "A streamlined temporal completness theorem" *Lecture Notes in Computer Science* vol 440, Springer Verlag 1990, pp. 322-336.

[50] Pinson L. and Wiener R. "An introduction to object-oriented programming and smalltalk". MA: Addison-Wesley Publishing Company, Reading, 1988.

[51] Pnueli A. "The temporal logic of programs". *19th Annual Symposium of Foundations of Computer Science*. IEEE, pp 46-57, 1977.

[52] Pnueli A. "The temporal semantics of concurrent programs". *Proceedings of the Symposium on Semantics of Concurrent Computation*, Lectures notes in Computer Science. 70, pp. 1-20, 1979.

[53] Pnueli A. and Harel E. "Application of temporal logic to the specification of real-time systems. In *Formal techniques in real-time and fault tolerant systems*, Lectures Notes in Computer Science, vol 331, pp. 84-98. Berlin: Springer-Verlag, 1988.

[54] Rasure J., and Young M. "Data flow visual languages". *IEEE Potentials*, vol. 11, no. 2, pp. 30-33, April, 1992.

[55] Rescher N. and Urquhart A. *Temporal Logic*. Springer-Verlag, 1971.

[56] Rubin R., Walker J., and Golin E. "Early experience with the Visual Programmer's WorkBench". *IEEE Transactions on Software Engineering*, vol. 16, no. 10, pp. 1107-1120, October 1990.

[57] Rus T. "Technical Report". *Computer Science*. Iowa University, May 1992.

[58] Sisirucá A. and Ionescu D. "A visual programming environment for a temporal logic language". *IEEE Canadian Conference in Electrical and Computer Engineering*, vol. 1, pp. 245-248, September 1993.

[59] Stoyenko A. "A schedulability analyzer for real-time euclid". In *Proceedings on Real-Time Systems Symposium*, The Computer Society of IEEE, pp. 218-227, September 1987.

[60] Tanimoto S. "The elements of artificial intelligence: An introduction to LISP", Computer Science Press, Maryland, 1987.

[61] Thistle J. and Wonham W. "Control Problems in a temporal logic framework. *International Journal of Control*, vol. 4, pp. 943-976, 1986.

[62] Xudong H. "Specifying and verifying real-time systems using time Petri nets and real-time temporal logic". *Proceedings of the 6th Annual Conference on Computer Assurance. Compass '91*, pp. 135-140, 1991.

[63] Zeigler B. "Hierarchical modular discrete event modeling in an object oriented environment". *Simulation 49*, vol. 5, pp. 219-230, 1987.

[64] Zeigler B. "Object-oriented simulation with hierarchical modular models". San Diego: Academic Press, 1990.

[65] * * *. "Objective-C ICpak201 user interface classes, reference manual, Stepstone Corporation, 1989

[66] * * *. "Special Issue on Dynamics of Discrete Event Systems", *Proceedings of IEEE*, vol.77, no.1, 1989.

Appendix A

Graphical Format of a Rule

Following is the format in which every rule of the system is stored in the file that contains the graphical information about the rules. The meaning is written beside each symbol.

```
$$              a new rule starts here
&LogicalCls     the next icon is of class LogicalCls.
AND( )          the name of the icon.
+354            the X coordinate of the icon on the screen.
-350            the Y coordinate of the icon on the screen.
<               the list of icons that will be connected to the
                logical operator created above starts here.
#PredicateCls   the next icon is of class PredicateCls.
PROCESS         predicate name.
+178            X coordinate.
-212            Y coordinate.
=0              lower time bound.
*0              upper time bound.
@0ArgumentCls   first argument of the above predicate.
```

```
SENDERX          name of the first argument.
+46              X coordinate.
-196             Y coordinate.
@1ArgumentCls    second argument of the above predicate.
RECEIVER0        name of the second argument.
+50              X coordinate.
-281             Y coordinate.
#PredicateCls    second icon of class predicate.
RECEIVED         predicate name.
+167             X coordinate.
-467             Y coordinate.
=1               lower time bound.
*4               upper time bound.
!TemporalCls     the next icon is of class TemporalCls.
UNTIL            name of temporal operator.
+194             X coordinate.
-388             Y coordinate.
@0ArgumentCls    first argument of the above predicate.
RECEIVER         name of the first argument.
+46              X coordinate.
-448             Y coordinate.
@1ArgumentCls    second argument of the above predicate.
PACKET0          name of the second argument.
+44              X coordinate.
-533             Y coordinate.
#PredicateCls    third icon of class predicate.
LESSTHAN         predicate name.
+216             X coordinate.
-634             Y coordinate.
```

```
=0              lower time bound.
*0              upper time bound.
@0ArgumentCls   first argument of the above predicate.
1               name of the first argument.
+84             X coordinate.
-623            Y coordinate.
@1ArgumentCls   second argument of the above predicate.
4               name of the second argument.
+89             X coordinate.
-710            Y coordinate.
>               end of items connected to logical operator.
#Conclusion     next it an icon of class Conclusion
PROCESS         name of the conclusion.
+569            X coordinate.
-380            Y coordinate.
=0              lower time bound.
*0              upper time bound.
@0ArgumentCls   first argument of the above predicate.
SENDER0         name of the first argument.
+427            X coordinate.
-431            Y coordinate.
@1ArgumentCls   second argument of the above predicate.
RECEIVER1       name of the second argument.
+423            X coordinate.
-476            Y coordinate.
-               end of rule.
```

Appendix B

Graphical Format of a Fact

#FactCls	an icon of class FactCls.
PROCESS	name of predicate of the fact.
+286	X coordinate.
-135	Y coordinate.
=0	lower time bound.
*0	upper time bound.
@0ArgumentCls	first argument.
SENDERX	name of first argument.
+88	X coordinate.
-117	Y coordinate.
@1ArgumentCls	second argument.
RECEIVER0	name of second argument.
+87	X coordinate.
-182	Y coordinate.
~0.1	cost value

Appendix C

Graphical Format the Database of Predicates

The following are all the predicate names that were created for the simulation of the three examples presented throughout this thesis. These examples are: the Alternating Bit Protocol, the Packet-Switched Communication Network, and the Telephone System. This is the graphical file, the text string after the sign "#" is the predicate name and the decimal digit beneath is the number of argument the predicate can handle:

```
#TIMEOUT        predicate name.
>2               number of arguments handled.
#PROCESS
>2
#LOST
>2
#RECEIVED
>2
#IS
>2
```

```
#GETS
>2
#RINGS
>2
#RINGSANDIS
>2
#DIALS
>2
#REQUESTS
>2
#RESPONDED
>2
#CONFIRMED
>2
#EXPIRES
>2
#BEGINS
>2
#FINISHES
>2
#ISNOT
>2
#ISPUT
>2
#GREATERTHAN
>2
#LESSTHAN
>2
```

Appendix D

Events for the examples.

D.1 Packet-Switched Communication Protocol

$e_1 = REQUESTS[NODE1CONNECTION]1.0;$

$e_2 = REQUESTS[NODE2CONNECTION]2.0;$

$e_3 = RESPONDED[NODE1CALL]3.0;$

$e_4 = RESPONDED[NODE2CALL]4.0;$

$e_5 = CONFIRMED[NODE1CONNECTION]1.0;$

$e_6 = CONFIRMED[NODE2CONNECTION]2.0;$

$e_7 = EXPIRES[NODE1CONNECTION]3.0;$

$e_8 = EXPIRES[NODE2CONNECTION]4.0;$

$e_9 = FINISHES[NODE1TRANSMISSION]5.0;$

$e_{10} = FINISHES[NODE2TRANSMISSION]6.0;$

$e_{11} = BEGINS[NODE1TRANSMISSION]1.0;$

$e_{12} = BEGINS[NODE2TRANSMISSION]2.0;$

D.2 Packet-Switched Communication Protocol

$e_1 = IS[PHONE1PICKEDUP]1.0;$

$e_2 = IS[PHONE2PICKEDUP]2.0;$

$e_3 = IS[PHONE1HUNGUP]3.0;$

$e_4 = IS[PHONE2HUNGUP]4.0;$

$e_5 = GETS[PHONE1ANSWER]1.0;$

$e_6 = GETS[PHONE2ANSWER]2.0;$

$e_7 = DIALSANDRINGS[PHONE1PHONEX]3.0;$

$e_8 = DIALSANDRINGS[PHONE2PHONEX]4.0;$

$e_9 = RINGSANDIS[PHONE1PICKEDUP]5.0;$

$e_{10} = RINGSANDIS[PHONE2PICKEDUP]6.0;$

$e_{11} = DIALS[PHONE1BUSY]1.0;$

$e_{12} = DIALS[PHONE2BUSY]2.0;$

Appendix E

General Architecture of the VPE.

E.1 Architecture of the VPE.

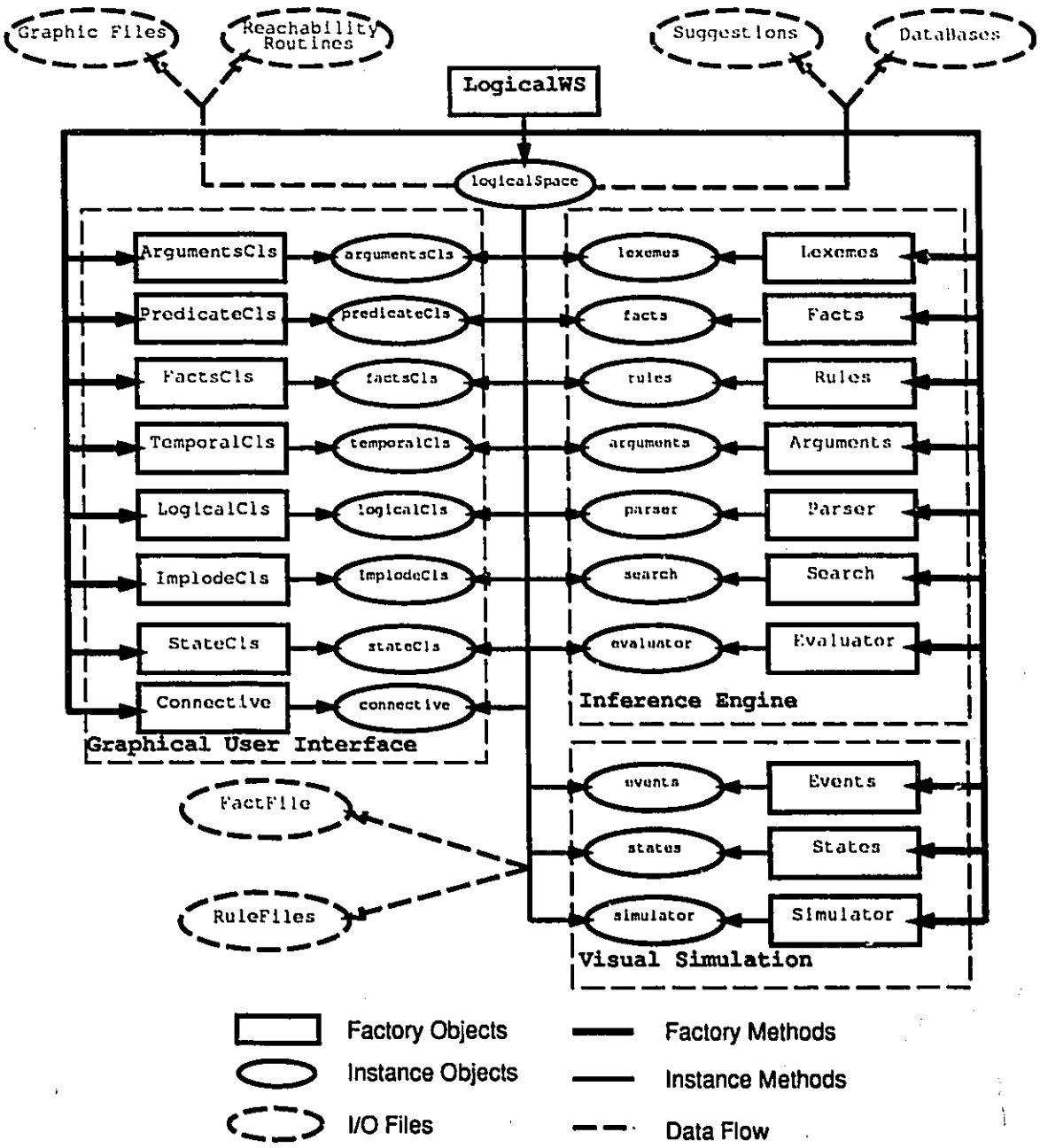

Figure E.1: General Architecture.