Spring 1998

A Very High Level Logic Synthesis

Norbert Ange Valverde
University of New Hampshire, Durham

A Very High Level Logic Synthesis

BY

Norbert A. VALVERDE

Master of Science in Automation and Micro-Electronics

Submitted to the University of New Hampshire
in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

in

Engineering

May 1998

This dissertation has been examined and approved.

[illegible]

Dissertation director, Andrzej Rucinski
Professor of Electrical and Computer Engineering

[illegible]

Jean-François Santucci
Professor of Electrical and Computer Engineering, Université de Corse, FRANCE

David J Forrest

David Forrest
Associate Research Professor of Space Science Center

Filson Glanz

Filson Glanz
Professor of Electrical and Computer Engineering

[illegible]

Richard Messner
Associate Professor of Electrical and Computer Engineering

[illegible]

John Pokoski
Professor of Electrical and Computer Engineering

1/20/99

Date

Dedication

I dedicate this dissertation to:

- my parents who considered the education of my sisters and me as a top priority:
- my niece Audrey to whom I wish to understand the value of education:
- my sisters Véronique and Corinne who were always there to help me out;
- and, most of all, my lovely wife Françoise who has been very supportive all along my thesis work and was willing to accept the challenges of changing her life style for me.

Je dédie cette thèse à:

- *mes parents qui ont placé mon éducation ainsi que celle de mes soeurs avant tout autre chose;*
- *ma nièce Audrey à qui je souhaite de comprendre la valeur d'une bonne éducation;*
- *et, avant tout, ma tendre épouse Françoise qui m'a soutenu, encouragé tout au long de ma thèse et qui a accepté le défit de changer de mode de vie pour moi.*

Acknowledgments

This dissertation has been possible thanks to the supports and advises of Dr. Jean-François Santucci and Dr. Andrjez Rucinski. Their friendship gave me the opportunity to experience the US life style as well as its working environment.

I acknowledge the precious financial support from EERIE (Ecole pour les Etudes et la Recherche en Informatique et Electronique), Nîmes France under the direction of Jean-Claude Ippolito.

I want to thank Dr. David Forrest, Dr. Filson Glanz, Dr. Richard Messner and Dr. John Pokoski for accepting to be in my thesis committee.

While being student at UNH, the encouragement and friendship of Brad Gillespie, Karen Hein, Andrew Kun and Tariq Nazeer helped me tremendously. They, all, contributed to the success of my integration at UNH and the completion of my PhD.

Table of Contents

List of Tables

List of Figures

List of Symbols

$cond_{s_i}^{s_j}(p, i)$	Transition condition predicate i.e. the transition condition predicate from state s_i to s_j ($s_i, s_j \in S$) is true when p is true at the instant $i \in \Xi$
d_{bhv}	Behavioral domain
d_{phl}	Physical domain
d_{str}	Structural domain
d_{λ}	Distance of an evolution process
$e(p, i)$	Event predicate i.e. the event predicate is true when a proposition p holds true at the instant $i \in \Xi$
ℓ_{act}	Architecture Level
ℓ_{cct}	Circuit Level
ℓ_{cpt}	Concept Level
ℓ_{imp}	Implementation Level
ℓ_{lgc}	Logic Level
ℓ_{sys}	System Level
p	Proposition defined as a statement which can be significantly characterized as either true or false

A	Set of attributes such that $a \in A$, a being an attribute
A^*	Equivalence class quotient of A under $R1$ ($A^* = A/R1$)
A_{ext}	Set of attributes such that $A \subset A_{ext}$ and additional elements of A_{ext} are attributes derived from L_{ext}
$A_{ext}{}^*$	Equivalence class quotient of A_{ext} under $R1_{ext}$ ($A^*_{ext} = A_{ext}/R1_{ext}$)
C	Set of characteristics such that $c \in C$, c being a characteristic
C^*	Equivalence class quotient of C under $R2$ ($C^* = C/R2$)
C^*_{ext}	Equivalence class quotient of C_{ext} under $R2_{ext}$ ($C^*_{ext} = C_{ext}/R2_{ext}$)
C_{ext}	Set of characteristics such that $C \subset C_{ext}$ and additional elements of C_{ext} are characteristics derived from L_{ext}
D	Description domains in design space DS such that $D = \{d_{bhv}, d_{str}, d_{phl}\}$
DS	Design Space $DS = < D, L, A, C, \delta, \lambda >$
EDS	Extended Design Space $EDS = < D, L_{ext}, A_{ext}, C_{ext}, \delta_{ext}, \lambda_{ext} >$
$F(p, I)$	Fact predicate i.e. the fact predicate is true when a proposition p holds true over the time interval $I \in \Upsilon$
GTS	Global Time Set i.e. $GTS = \Xi \cup \Upsilon$
I_d	Time interval defining the life-time of a system state
L	Partially ordered class of abstraction levels $< L, <_L >$ in design space DS such that $L = \{\ell_{imp}, \ell_{cct}, \ell_{lgc}, \ell_{act}, \ell_{sys}\}$
L_{ext}	Extended levels of abstraction in EDS such as: $L_{ext} = L \cup \{\ell_{cpt}\}$
$M_{zeigler}$	Model capturing the pseudo-state diagram using the formalism of Zeigler such as: $M_{zeigler} =< S, I, F, O, \tau_{int}, \tau_{ext}, \psi_z, t_a, \Gamma >$

$R1$	Attribute equivalence relation over A defined as $\forall x, y \in A, x\ R1\ y \Leftrightarrow \rho(x) = \rho(y)$
$R2$	Characteristic equivalence relation over C defined as $\forall x, y \in C, x\ R2\ y \Leftrightarrow \zeta(x) = \zeta(y)$
$R1_{ext}$	Extended attribute equivalence relation over A_{ext} defined as $\forall x, y \in A_{ext}$, $x\ R1_{ext}\ y \Leftrightarrow \rho_{ext}(x) = \rho_{ext}(y)$
$R2_{ext}$	Extended characteristic equivalence relation over C_{ext} defined as $\forall x, y \in C_{ext}, x\ R2_{ext}\ y \Leftrightarrow \zeta_{ext}(x) = \zeta_{ext}(y)$
S	Set of state of the pseudo-state diagram
$STATE(I, s_i)$	State predicate i.e. the state predicate is true when the system is in state $s_i \in S$ during the time interval $I \in \Upsilon$
T	Time Space
t_a	Duration of the life-time of a system state
V_d	Set of descriptive variables such that $V_d = V_i \cup V_{n-i}$
V_i	Set of input variables
V_s	Set of state variables
V_{n-s}	Set of none-state variables
V_{n-i}	Set of none-input variables such that $V_{n-i} = V_s \cup V_{n-s}$
$\Re$	Real numbers
$\aleph$	Natural numbers
δ	Function such as $\delta : D \times L \rightarrow A^* \times C^*$ maps a description domain from D and a level of abstraction from L onto an attribute set from A^* and a

	characteristic set from C^* (at each point in DS, a set of attributes and characteristics can be extracted)
δ_{ext}	Function such as $\delta_{ext} : D \times L_{ext} \rightarrow A^*_{ext} \times C^*_{ext}$ maps a description domain from D and a level of abstraction from L_{ext} onto an attribute set from A^*_{ext} and a characteristic set from C^*_{ext}
ζ	Property such as $\zeta : C \rightarrow D \times L$ maps each characteristic $c \in C$ onto a description domain from D and a level of abstraction from L
ζ_{ext}	Property such as $\zeta_{ext} : C_{ext} \rightarrow D \times L_{ext}$ maps each characteristic of C_{ext} onto a description domain from D and a level of abstraction from L_{ext}
λ	Evolution process defined as $\lambda : D \times L \rightarrow D \times L$
$\lambda_{cpt-reft}$	Concept refinement evolution defined as $\lambda_{cpt-reft}(d_{bhv}, \ell_{cpt}) = (d_{str}, \ell_{cpt})$
$\lambda_{cpt-synt}$	Concept synthesis evolution defined as $\lambda_{cpt-reft}(d_{str}, \ell_{cpt}) = (d_{bhv}, \ell_{sys})$
λ_{ext}	Evolution process defined as $\lambda_{ext} : D \times L_{ext} \rightarrow D \times L_{ext}$
λ_u	Unary evolution process with a distance of 1
ρ	Property such as $\rho : A \rightarrow D \times L$ maps each attribute $a \in A$ onto a description domain from D and a level of abstraction from L
ρ_{ext}	Property such as $\rho_{ext} : A_{ext} \rightarrow D \times L_{ext}$ maps each attribute of A_{ext} onto a description domain from D and a level of abstraction from L_{ext}
τ_{ext}	External transition function such that $\tau_{ext} = \tau_z$
τ_{int}	Internal transition function such that $\tau_z : S \rightarrow S$
τ_z	Transition function such that $\tau_z : S \times V_i \rightarrow S$
ψ_z	Output function such that $\psi_z : S \times V_i \rightarrow V_{n-s}$

Γ	Priority function such that $\Gamma : S \times S \rightarrow \aleph$
Ξ	Instantaneous time interval
Υ	Time interval
$<_L$	Partial ordering on L such as $\forall x, y \in L, x <_L y \Leftrightarrow$ x is less abstract than y
$\#$	Cardinal of a set
$\forall$	For all
$\exists$	There exists
$\Rightarrow$	If ... then ...
$\Leftrightarrow$	If and only if
$\vee$	Or
$\wedge$	And
$\neg$	Not
$\cap$	$\cap B$= the intersection of all members of B
$\cup$	$\cup B$= the union of all members of B
$\times$	$A \times B$= Cartesian product of A and B

Note on typographical styles:

italic	Introduction of a new concept or term
bold/italic	Emphasis a group of words which is under consideration in a defintion, theorem or corollary
UNDERLINE CAPITAL	reserved words in VHDL
□	End of proof

$(\ldots)$	Ordered elements
$\{\ldots\}$	Set
$< \ldots >$	Definition of a model

List of Acronyms

AI	Artificial Intelligence
ASIC	Application Specific Integrated Circuit
ASIM	Application Specific Integrated Module
ASM	Algorithmic State Machine
ATPG	Automatic Test Pattern Generator
ATW	Advanced Technology Workshop
BSDL	Boundary Scan Description Language
CAD	Computer Aided Design
CAE	Computer Aided Engineering
CATSAT	Cooperative Astrophysics and Technology SATellite
CEEDA	Collaborative Engineering and Electronic Design Automation Conference
CPLD	Complex Programmable Logic Device
CSP	Communication Sequential Process
DSP	Digital Signal Processing
EDA	Electronic Design Automation

EERIE	Ecole pour les Etudes et la Recherche en Informatique et Electronique
ESDA	Electronic System Design Automation
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
HLS	High Level Synthesis
KRS	Knowledge Representation System
MCM	Multi-Chip Module
PCB	Printed Circuit Board
RAM	Random Access Memory
RT	Register Transfer
SDL	Specification and Description Language
SPECIAL	Specification Procedure for Electronic Circuits in Automation Language
SRAM	Static Random Access Memory
UNH	University of New Hampshire
VHDL	VHSIC Hardware Description Language
VHLLS	Very High Level Logic Synthesis
VHSIC	Very High Scale Integrated Circuit
VLSI	Very Large Scale Integration

ABSTRACT

A Very High Level Logic Synthesis

by

Norbert A. VALVERDE
University of New Hampshire, May, 1998

The evolution of Computer Aided Design (CAD) calls for the incorporation of design specifications into a microelectronics system development cycle. This expansion requires the establishment of a new generation of CAD procedures, defined as *Very High Level Logic Synthesis (VHLLS)* . The fundamental characteristics of open-ended VHLLS are: (1) front-end graphical interface; (2) time encapsulation; and (3) automatic translation into a behavioral description. Consequently, the VHLLS paradigm represents an advanced category of CAD-based microelectronics system design, built on a deep usage of expert systems and intelligent methods. Artificial Intelligence (AI) formalisms such as Knowledge Representation System (KRS) are necessary to model properties related to the very high level of specification such as: dealing with ambiguities and inconsistencies, reasoning, computing high-level specification, etc. A prototype VHLLS design suite, called *Specification Procedure for Electronic Circuits in Automation Language (SPECIAL)* , is defined, compared with today's commercial tools and verified using numerous design examples. As a result, a new family of formal and accelerated development methodologies has become feasible with a better understanding of formalized knowledge driving these design processes.

Chapter 1

Introduction

The evolution of automatic design methodologies is moving to new ground. Understanding this evolution is the primary purpose of this thesis. In addition, it is proposed a view to characterize the next generation of automatic design methodologies.

The influence of automation in the design space of microelectronics systems is affecting a larger scope of risky and uncertain design decisions than before. At the same time no single de facto strategic direction in design methodologies appears to be emerging, reflecting the reality that the design is as much art as engineering. This chapter characterizes some directions in which the design of microelectronics systems is evolving, with a new formal representation of the design space and associated automation procedures selected as a gradient in advancing the knowledge about design processes.

1.1 Research Problem Statement

1.1.1 Research Motivation

The complexity of a design cycle governs the strategy undertaken by an electronic system

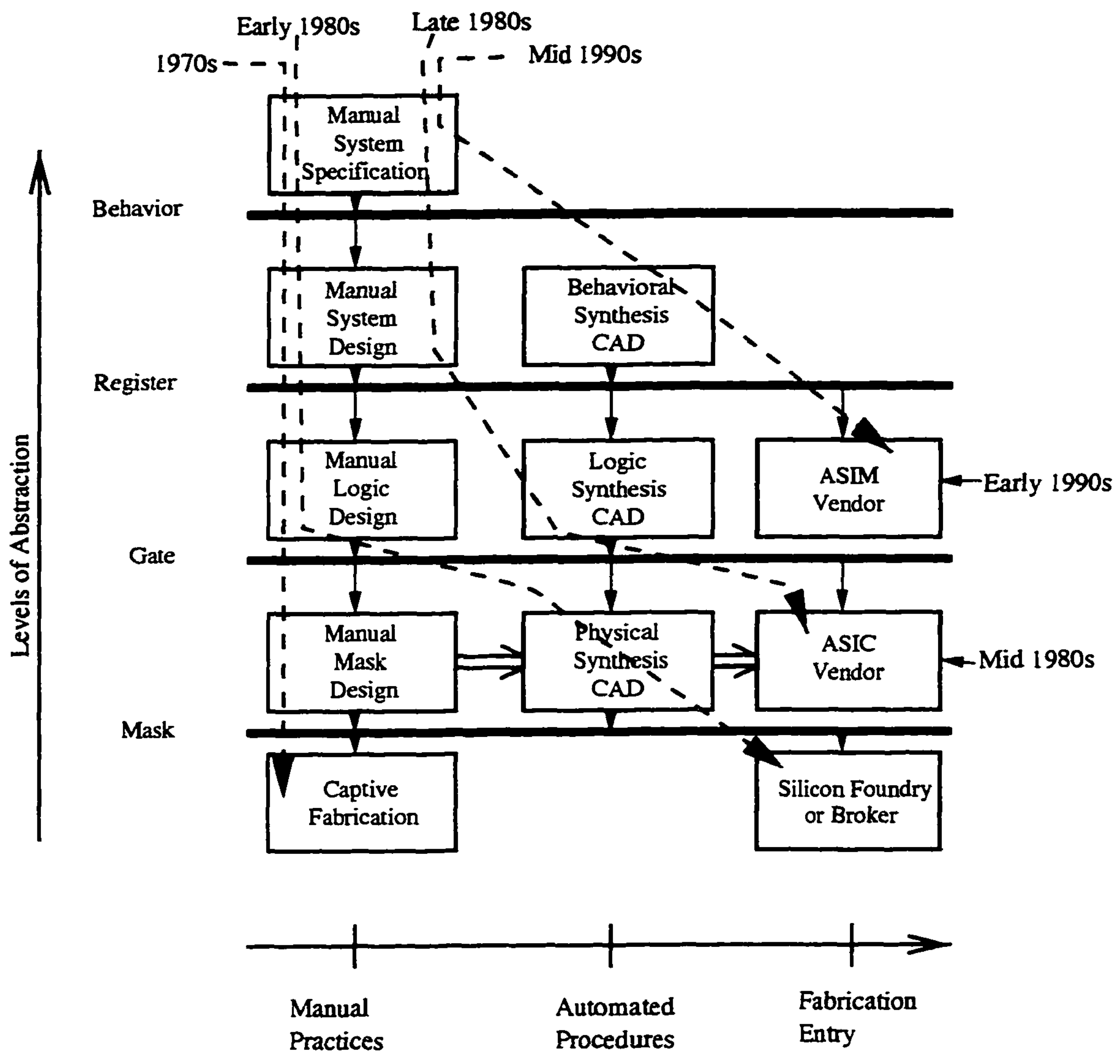


Figure 1-1: Status of Electronic Design Technologies

developer. A common approach, the *top-down approach*, is to start with a more abstract description when the complexity of the system is higher. Fig. 1-1 shows the evolution of design practices over time highlighting that as complexity increased, design tools were developed to define a system at a higher level of abstraction [NEW91]. In the 1970s, it was common practice to design the whole system manually from the system level description to the fabrication of Printed Circuit Boards (PCBs) . In the early 1980s, some Computer Aided Design (CAD) tools were promoting *physical synthesis* to take over the manual mask design task. Physical synthesis from one library to another on the logic level, also referred to as technology mapping, is accomplished by deriving the behavioral description in terms of Boolean expressions, and resynthesizing it with a new library. Silicon compilation [BCM+88] is a member of this physical synthesis class. In the meantime, a new type of component emerged which was based on silicon. In the mid-1980s, Application Specific Integrated Circuit (ASIC) components became a valuable option for designers and increased the need for more automation in the electronics design flow. In late 1980s and early 1990s, *logic synthesis* emerged as an alternative to manual logic design which could also support other applications specialized in verification, test, libraries, etc ..., tasks which are time consuming and cumbersome. Logic synthesis translates Boolean expressions into a netlist of components from a given library of logic gates such as NAND, NOR, XOR, etc .... Within the past two years, ***behavioral synthesis***, also referred to as ***register-transfer (RT) synthesis***, has gained in popularity in CAD systems [BLA97]. RT synthesis starts with a set of states and a set of register-transfers in each state. One state corresponds roughly to a clock cycle. Register-transfer synthesis generates the corresponding structure in two parts: (a) a datapath which emphasizes data processing and (b) a unit control responsible for control signal scheduling. Application Specific Integrated Module (ASIM) components such as Field Programmable Gate Arrays (FPGAs) are highly dependent on this synthesis process to confine the design complexity. The design space discussed above is concisely encapsulated in Fig. 1-2 (also known as the Y diagram) [DGLW92] as we will discuss more thoroughly in Chapter 3.

The analysis of the evolution of electronics design technologies clearly indicates that from the 1970s the behavioral level is the highest level of abstraction in design automation commonly accepted as an entry level. Typically:

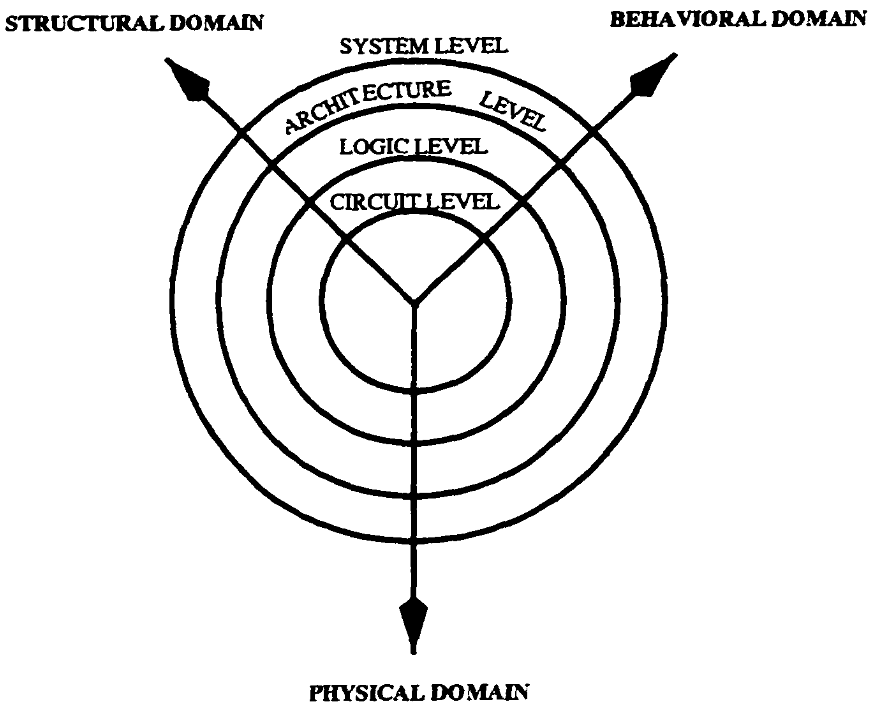


Figure 1-2: Y Diagram

- The microelectronics industry uses the RT level as its highest level of abstraction to initiate a design process. This entry level is commonly offered by CAD vendors such as Mentor Graphics^{TM} [Cor95], Viewlogic^{TM} [Inc97], and others;
- Research (contrary to the above) is focused on *High Level Synthesis (HLS)* to translate a behavioral description into the RT level. HLS is the transformation of a behavioral description into a set of connected storage and functional units. Typically, the types of algorithm generally used in HLS are partitioning, scheduling, and allocation [DGLW92, AB94].

Note that this observation is consistent with a notion of research preceding the availability of commercial tools.

Another aspect of design is the complexity of tools used to facilitate the design flow. On one hand, tools alleviate certain steps such as the interpretation of a symbol as its corresponding physical representation, the placement and routing processes of components, etc. On the other hand, design needs to take into consideration a multiplicity of aspects, typically each with a separate CAD tool. Therefore, for a single design, a suite, rather than one complex tool is used. For example, during a Multi-Chip Module (MCM) device design (further information on MCM design can be found in [HEI95, JIA95]), four main families of tools are involved in a Mentor Graphics^{TM} design environment [Cor93b]:

- Capture of the system description using VHSIC Hardware Description Language (VHDL) (a text editor and the package sys_1076^{TM} to compile VHDL design files);
- Synthesis of the VHDL code into a hierarchical structure with the top level containing symbols of dies which are mounted on the substrate of an MCM device (Autologic^{TM});
- Design of each die using a Very Large Scale Integration (VLSI) method (IC Station^{TM}); and finally
- Preparation of the MCM device for fabrication (MCM Station^{TM}).

The total number of tools involved during this design task is 11 as illustrated in Fig. 1-3. Such mutation requires sophisticated training which emphasizes the tools' functionalities instead of focusing on new design techniques and technologies.

In summary, there is an acute need to start the design process at a highest possible level of abstraction (as part of a natural evolution of the CAD methodologies). This evolutionary step is driven by the mutation of the electronic design world where miniaturization [KAT82] and system-on-chip [KM91] are continuously sought. It must manage or even reduce, the complexity of new design processes. These statements are seconded by a quotation:

"In order to move upward efficiently, we need to build other languages *on top of*

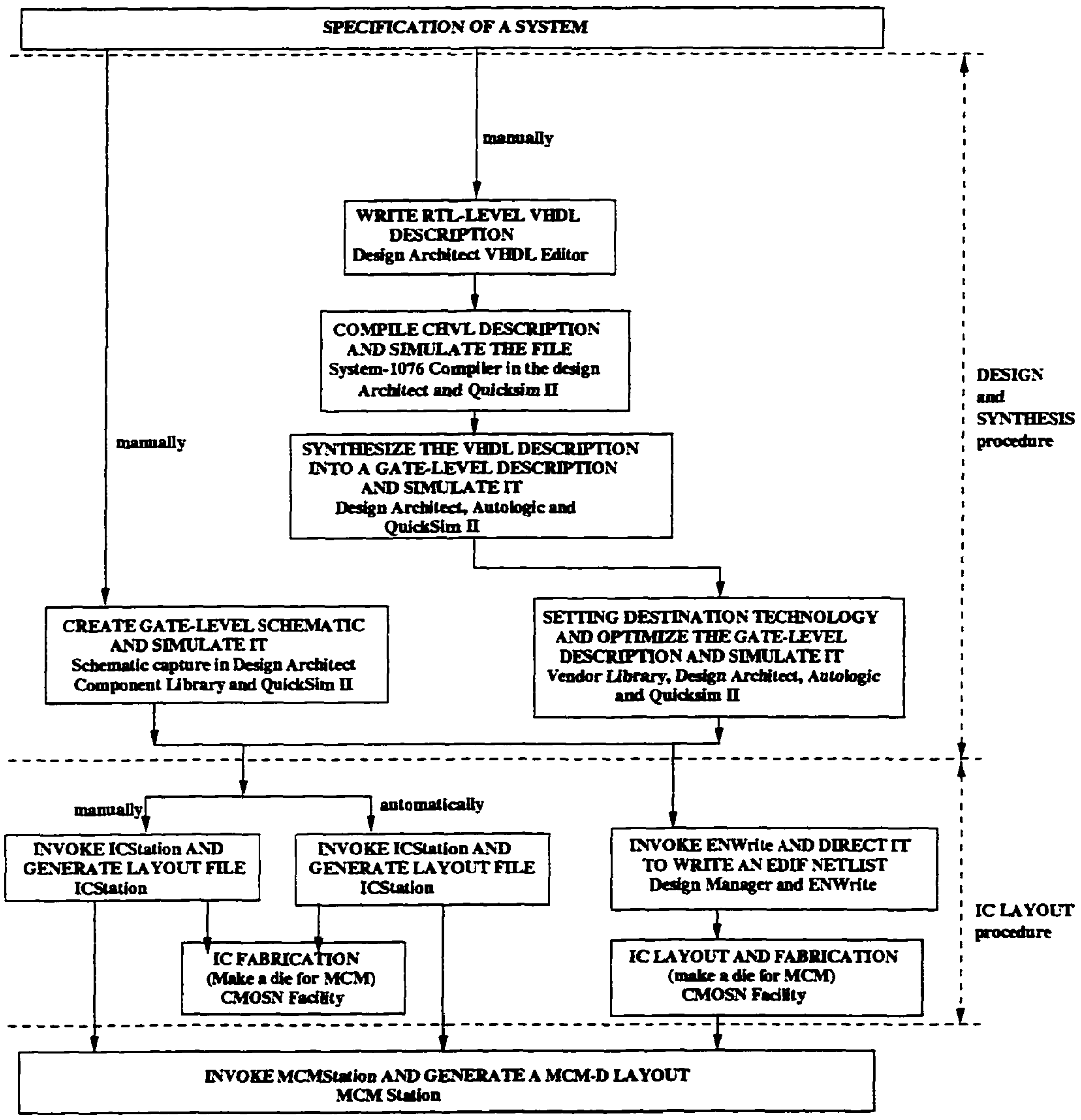

SPECIFICATION OF A SYSTEM
manually
WRITE RTL-LEVEL VHDL DESCRIPTION Design Architect VHDL Editor
COMPILE CHVL DESCRIPTION AND SIMULATE THE FILE System-1076 Compiler in the design Architect and Quicksim II
SYNTHESIZE THE VHDL DESCRIPTION INTO A GATE-LEVEL DESCRIPTION AND SIMULATE IT Design Architect, Autologic and QuickSim II
manually
CREATE GATE-LEVEL SCHEMATIC AND SIMULATE IT Schematic capture in Design Architect Component Library and QuickSim II
SETTING DESTINATION TECHNOLOGY AND OPTIMIZE THE GATE-LEVEL DESCRIPTION AND SIMULATE IT Vendor Library, Design Architect, Autologic and Quicksim II
DESIGN and SYNTHESIS procedure
manually
automatically
INVOKE ICStation AND GENERATE LAYOUT FILE ICStation
INVOKE ICStation AND GENERATE LAYOUT FILE ICStation
INVOKE ENWrite AND DIRECT IT TO WRITE AN EDIF NETLIST Design Manager and ENWrite
IC FABRICATION (Make a die for MCM) CMOSN Facility
IC LAYOUT AND FABRICATION (make a die for MCM) CMOSN Facility
IC LAYOUT procedure
INVOKE MCMStation AND GENERATE A MCM-D LAYOUT MCM Station

Figure 1-3: Mentor Graphics^{TM}: Design Tasks

VHDL to represent familiar concepts used by systems designers", from DUTT et al. in "High-level synthesis : introduction to chip and system design" [DGLW92]

1.1.2 Hypothesis

To meet the design challenge outline in the previous section, this research activity is envisioned to:

1. Introduce a new design process using a generalized synthesis approach as shown in Fig. 1-4. The emphasis in this thesis is on the front-end synthesis, called Very High Level Logic Synthesis (VHLLS);

2. Introduce the next generation of design automation tools as a practical consequence of a generalized synthesis process;

3. Lessen or at least maintain the complexity of microelectronics systems design by starting a design process at a higher level of abstraction;

4. Incorporate a high-level specification as the entry level in an automated design flow.

Indeed, item (1) represents our primary objective. Item (2) introduces the issue of feasibility to this problem. Therefore, the hypothesis of the research problem can be stated as follows: having (1), we can define (2) or mathematically $((1) \Rightarrow (2))$. If $((1) \Rightarrow (2))$ is true then (3) and (4) are the properties of the new design methodology. In other words, (hypothesis $((1) \Rightarrow (2))) \Rightarrow ((3) \wedge (4))$ becomes a theorem.

In order to characterize the next generation of design automation tools, a list of desired properties [GVN93] is introduced. These properties are called *characteristics* of a design automation tool, and are dependent on the level of abstraction the tool is designed to work at. A set of characteristics is defined and denoted as C. This set should be a non-restrictive,

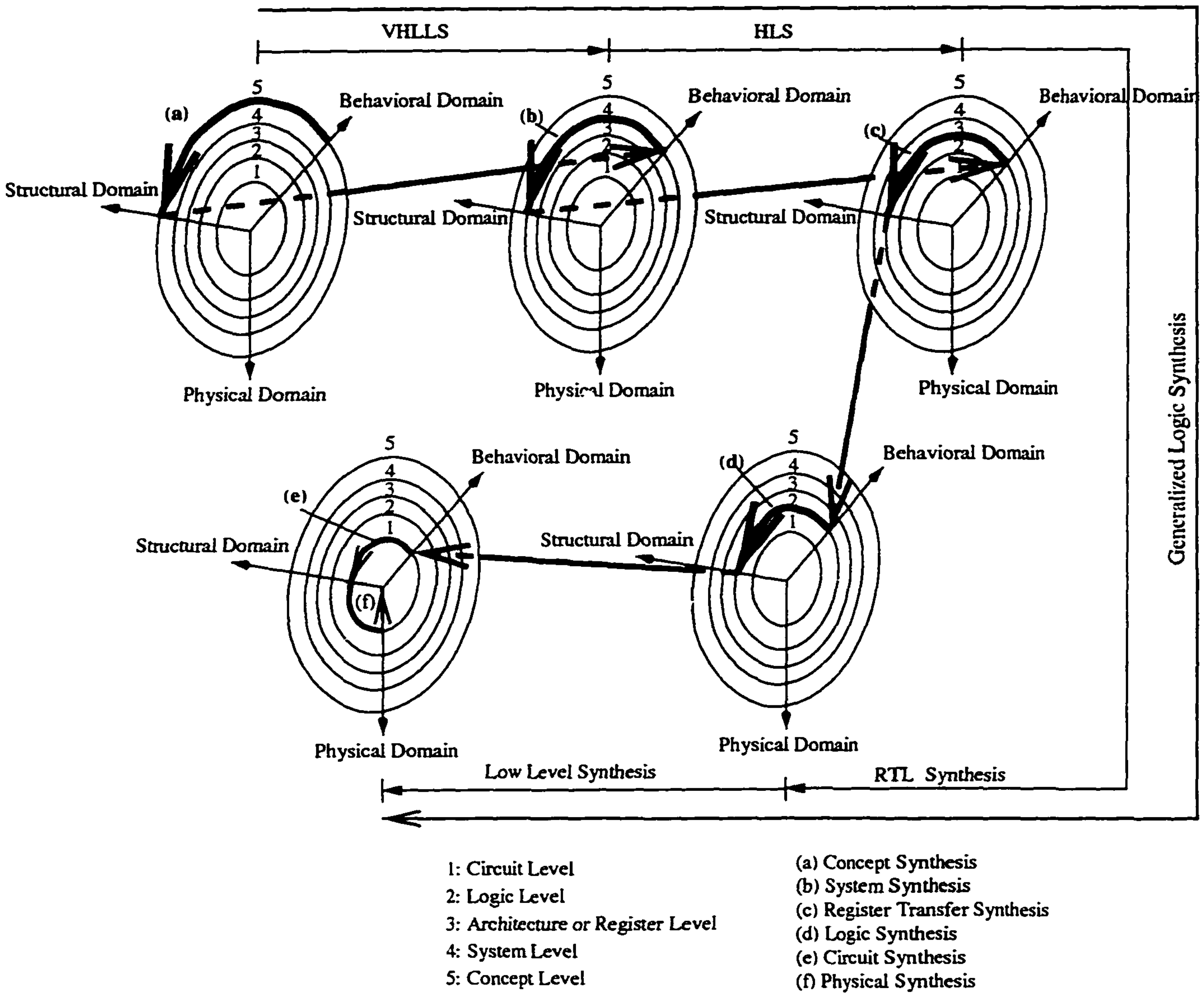


Figure 1-4: Theoretical Design Process

but, bounded set, to keep the problem tractable C is then defined as a set of a finite number of elements. So, the elements of C allow a classification of design automation tools which indicate their characteristics or peculiar qualities. For the next generation of CAD, a non-exhaustive proposed list of characteristics is as follows[1]:

[1]Note that the proposed schema does not preclude defining a different set of characteristics for the next generation of CAD tools.

- *Sequentially Decomposable Activities*: an action[2] can be decomposable in a sequence of sub-actions. The use of sequential actions is a common practice in engineering. When a system is defined, designers practice a one step reasoning i.e. when one action is performed enabling changes to a new known configuration of the system, the next action captures this new system configuration and brings the system into another one, and so on. An analogy of sequential action is any structured programming language such as C:

- *Concurrently Decomposable Activities*: an action[2] can be decomposable in sub-actions which can be applied concurrently. The use of concurrent actions allows the system to perform actions at the same time and independently from one to another. In this case, shared resources become the bottleneck of the system performance. An analogy is parallel programming;

- *State Transitions*: a system can be described as a set of states under which a transition function defines a state change. States are predefined configurations of a system. A transition is a mechanism of changing a predefined system configuration to another one. In engineering, the most common state transition mechanisms used are Moore[3] or Mealy[4] Finite State Machine (FSM) ;

- *Immediate Mode Change*: at any instant and any system status, a system has the ability to apply an operational mode change instantly. In most systems, some external

[2]Action or equivalently activity is a particular mode of system behavior. It may be a computation, which is possibly complex or time-consuming, or it may be recursively defined as a composition of sub-activities, where the sub-activities may be sequential or concurrent to one another.

[3]In the Moore FSM, the output value is depending only on the state of the FSM

[4]In the Mealy FSM, the output value depends on the transition and the input values of the FSM.

events must be treated instantaneously. A common use of this characteristic is when a system receives a reset command, it needs to react at once even though the system is in the middle of a computation. So, this characteristic treats exceptional events which needs immediate attention;

- *Activity Completion*: a system ends its current activity before starting a new one. Such mechanism is important when associated with sequential activities. In this case, an action must be completed before starting the next one. This is crucial when a designer uses a description language which mixes concurrent and sequential statements with no completion mechanism defined, this is the case with VHDL;

- *Delay Specification*: time constraints can be specified. When the time constraint elapses, the system changes its status automatically. This characteristic avoids the definition of a clock rate which is a critical constraint in a synchronous system. Therefore, the decision on the clock rate is then postponed until after the behavior of the system is validated. Indeed, if a designer needs to specify that the system under design must change its status after 40 ns, the ways of measuring these 40 ns are infinite i.e. a clock with a period of 25MHz can measure 40 ns as well as 50MHz, 75Mhz and any multiple of 25MHz. So, depending of the system, one clock rate may be better than another. There is no way to know the best fit before starting the system design;

- *Asynchronous Activities*: actions specified in a system do not depend on a global clock. These activities are reacting to an external change which are not correlated with any clocks. Interrupts are a good analogy to these asynchronous activities;

- *Design for {Testability, Manufacturability, etc}*: specific properties are added to the system to meet requirements for testability, manufacturing, etc. When a design is performed, the design methodology changes somewhat depending on the property to emphasize i.e. for space applications, a property to emphasize is the test and fault-tolerance because no failure is allowed while the spacecraft is in space, whereas in consumer electronics, the design for manufacturing property is emphasized to minimize cost;

- *Multiple Model Representations*: a system can be represented using a mixture of more than one description model. Depending on the type of system a designer wants to design, a unique description model might not be sufficient or appropriate to specify the full system. So, the use of appropriate description models will lead to a better description of a system;
- *Reusability*: a system or a sub-system is designed in such a way that it can be very easily reused for another projects. This characteristic is important to optimize the design process by reducing the time-to-market, eliminating repetitive activities, etc.

In summary, the acceptable design automation tool implementing the research vision outlined in Section 1.1.2 should conform to all the characteristics defined above. Table 1.1 recapitulates these characteristics:

C = {

Characteristics	Checkmark
Sequentially Decomposable Activities	✓
Concurrently Decomposable Activities	✓
State Transitions	✓
Immediate Mode Change	✓
Activity Completion	✓
Delay Specification	✓
Asynchronous Activities	✓
Design for { Testability, Manufacturing, etc }	✓
Multiple Model Representations	✓
Reusability	✓

}

Table 1.1: Acceptable Design Automation Tool Characteristics

It has to be stressed that in order to make the growing complexity of CAD tools tractable, not all the characteristics can be taken into consideration. A minimal configuration for the first generation of VHLLS is sufficient to demonstrate the feasibility of the proposed automatic process evolution. This first generation of VHLLS has been built upon an existing representation of the design space provided by the Y diagram. A new level of abstraction, called *concept level*, is introduced above the highest one defined in the Y diagram. This new level is indispensable in an attempt to formalize design specifications and their associated properties. So, VHLLS represents the synthesis process that links the concept level to the system level, as explained in detail in Chapter 3. This first generation of VHLLS has to automate the transition from the concept level to the system level with a tool able to conform to the following characteristics:

$C_{min} = \{$

Characteristics	Checkmark
Sequentially Decomposable Activities	✓
Concurrently Decomposable Activities	
State Transitions	✓
Immediate Mode Change	
Activity Completion	✓
Delay Specification	✓
Asynchronous Activities	
Design for { Testability, Manufacturing, etc }	
Multiple Model Representations	
Reusability	

$\}$

Table 1.2: First Generation VHLLS Characteristics

For the purpose of comparison, the most currently available advanced high-level synthesis tool, presented in Chapter 4, can be characterized such as:

$C_{SpecCharts} = \{$

Characteristics	Checkmark
Sequentially Decomposable Activities	✓
Concurrently Decomposable Activities	✓
State Transitions	✓
Immediate Mode Change	✓
Activity Completion	✓
Delay Specification	
Asynchronous Activities	
Design for { Testability, Manufacturing, etc }	
Multiple Model Representations	
Reusability	

$\}$

Table 1.3: Advanced High-Level Synthesis Tools Characteristics

The main difference between these two design methodologies is that the methodology characterized by C_{min} meets the requirement of "Delay Specification" as opposed to Spec-

Charts characterized by $C_{SpecCharts}$. Note that this set of characteristics gives a methodology to classify design procedures and to compare them. The first metric which can be applied is simply to use the cardinal of the characteristics set (noted #C) for a design procedure. For example, $\#C_{min} = 4$ or $\#C_{SpecCharts} = 5$ or $\#C = 10$. A method to classify these design procedures is to compare the cardinal of their characteristics set. So, from the above sets, we have:

$$\#C_{min} < \#C_{SpecCharts} < \#C$$

This means C_{min} verifies fewer characteristics than $C_{SpecCharts}$ and both of these methods do not meet the full requirement for the next generation of design automation tools. Other classification schemes can be considered by applying a weight coefficient to each element of the characteristics set. However, this issue goes beyond the scope of this thesis. Another issue related to the evolution of design automation tools can be characterized using the set of characteristics. A design automation tool is characterized by C_i. Its next generation can be characterized by C_{i+1} such as $\#C_i < \#C_{i+1}$.

1.1.3 Research Goals

To implement the hypothesis stated in the previous section, the research goals are formulated as follows:

1. The characterization of the design space along with a set of properties;
2. The formalization of a concept level in which high-level specifications are embedded;
3. The statement and formalization of an automatic process to migrate from the concept level to the system level which is called VHLLS and formally defined in Chapter 3 ;
4. The implementation of VHLLS, taking high-level specifications and translating them

into a behavioral description at the system level.

1.1.4 Merits and Contributions

A new design process is introduced at the front-end of the *generalized synthesis process* leading to the next generation of design automation tools. A characterization and formalization of the *design space* is necessary in order to bound the requirement of this new design process. As a consequence, a set of *characteristics* describing this new design process is introduced defining a metric to classify design automation tools. An unusual characteristic proposed among others is the *delay specification* which enables specification of time constraints for the system specifications independently from a clock. As a result of the introduction of this new design process, a *Concept Level* is defined as a new level of abstraction, above the system level. A formalism is proposed to represent the *evolution* of a design description in the design space. This synthesis process, called *VHLLS*, is introduced, enabling a link between the concept level and the behavioral level with a particular emphasis on time encapsulation. So far, commercial tools and research in design automation specify a global clock and use it to specify the remaining behavior of the system. A second approach can be taken considering that the measure of time is a very important constraint which must be fixed as late as possible in order to choose the best clock rate for the system under design. To do the above, a set of restrictions is taken into consideration to reduce the domain of investigation. As a result, some metrics have been defined. In addition, a feasibility study has been performed to expand the representation of the design space which led to VHLLS.

1.2 Thesis Organization

Chapter 2 describes the state of the art of the CAD domain in microelectronics. It emphasizes, among other things, the availability and sophistication of commercial CAD tools. Chapter 3 refines the new level of abstraction in the design process and adopts the design space accordingly. Chapter 4 provides an overview of relevant research in high-level synthesis and description styles for specification purposes. Chapter 5 discusses the VHLLS process and suggests two approaches to implement it. Chapter 6 presents a tool called SPECIAL which enables a designer to specify a system. A VHLLS process defined in the previous chapter can then be applied to generate a VHDL description automatically. In Chapter 7, three typical examples show the benefits and limits of a such approach. Finally, some conclusions and suggestions for future work in the area of VHLLS are proposed.

Chapter 2

CAD Domain in Microelectronics

Having stated the aim of the research activity, an overview of the concepts behind the term "CAD domain" is provided in this chapter along with a presentation of the status of commercial CAD tools. With these two fundamental elements, the principles of VHLLS are also stated.

2.1 Design Process Characterization

This section introduces the notions of design automation and design methodology. It is fundamental to understand the existential reason of these two notions in order to conceptualize the motivation and direction of this research. Currently, designers perceive that available design methodologies will soon become obsolete because of the rapid rise of system complexity. It should be stressed that this "unstable" phenomenon is typical for the whole CAD history (with no end in sight) leading then to a new design approach called the Electronic System Design Automation (ESDA) approach.

In the field of microelectronics system devices have become increasingly complex, reaching densities of millions of transistors per square centimeter. It has become more difficult to design such systems by handcrafting methods, that is, by representing each transistor or defining each signal in terms of logic gates. To manage the complexity, systems have to be designed at abstract levels where functionalities and tradeoffs are easier to comprehend. Design automation enables integrated circuit designers an opportunity to optimize design efforts at these levels with superior productivity and competitiveness. Furthermore, design automation empowers engineers with the ability to do rapid prototyping, consider mechanical and physical constraints, handle mixed-signal systems, etc. A consequence of this approach is the development of complex tools to automate the entire design process from concept to final implementation.

In the development of design automation methods and tools, a typical goal is to apply the concepts of (1) *first-silicon* and (2) *first-specification* [DGLW92] to reduce the time-to-market cycle for new devices. The first silicon concept is based on the principle that prototyping[1] is time consuming and costly. Traditionally prototyping is a critical stage because it allows verification of the system functions. The first silicon concept requires a design process where simulation prevails over prototyping during the validation stage of the finalized system. The simulation process uses *back-annotation*[2] to take into account the physical constraints of the circuit such as propagation delay, setting time, etc. Another important feature of the first silicon concept is automatic control of the physical design rules. Consequently, CAD tool assistance is crucial in verifying both functionality and design rules of the entire chip design cycle. The second concept, first-specification, has as its goal the reduction of the number of design iterations involved to just one. As opposed

[1]A prototype is viewed as a first physical realization of a design in order to check its behavior against its specifications.

[2]It is a method for importing low level timing informations to the level of description of which a system is captured in. Its purpose is to have a more realistic simulation of the design.

to the first-silicon concept, the first-specification concept requires accurate modeling of the design process and accurate estimation of the product's quality measurements such as performance and cost.

Today, commercial tools are mature enough to operate with relative accuracy at the level defined by first-silicon. Researchers and toolmakers are currently attempting to meet the challenge of the first-specification concept. To address these issues, there are two competing philosophies:

- top-down methodology ; and
- bottom-up methodology.

The *top-down* methodology, often referred to as "describe and synthesize", can be defined as a method for modeling a whole system using a high-level of abstraction. A synthesis process is applied to refine the system model into lower subsystems and lower abstraction levels closer to the target technology. The *bottom-up* methodology, often referred to as "capture and simulate", is a method for modeling a system starting with the lowest modules of the system hierarchy, and building the whole system using a combination of these modules. Simulation is performed on each module to ensure proper functionality. Modules are combined to form larger modules, creating new levels in the system hierarchy. The level of hierarchy terminates when a combination of modules reaches the top system level.

Currently, available tools promote the use of the bottom-up methodology when the description entry is a schematic form. With the emergence of Hardware Description Languages (HDLs) [3] at the entry level, the top-down methodology becomes more effective. Often prominent toolmakers such as Mentor Graphics^{TM} or Viewlogic^{TM} combine both

[3]HDLs are like programming languages but specialized in the description of microelectronics hardware.

methodologies in their tool set. For example, during the design process of an FPGA, the design starts with a VHDL description which promotes top-down methodology. Because the target technology is FPGA, designers must use a library of components and practice bottom-up methodology. The practical result therefore is a description with both methodologies mixed together. When a description is technology independent, the top-down methodology is the most appropriate. It is even more appropriate when description entries are high-level specifications.

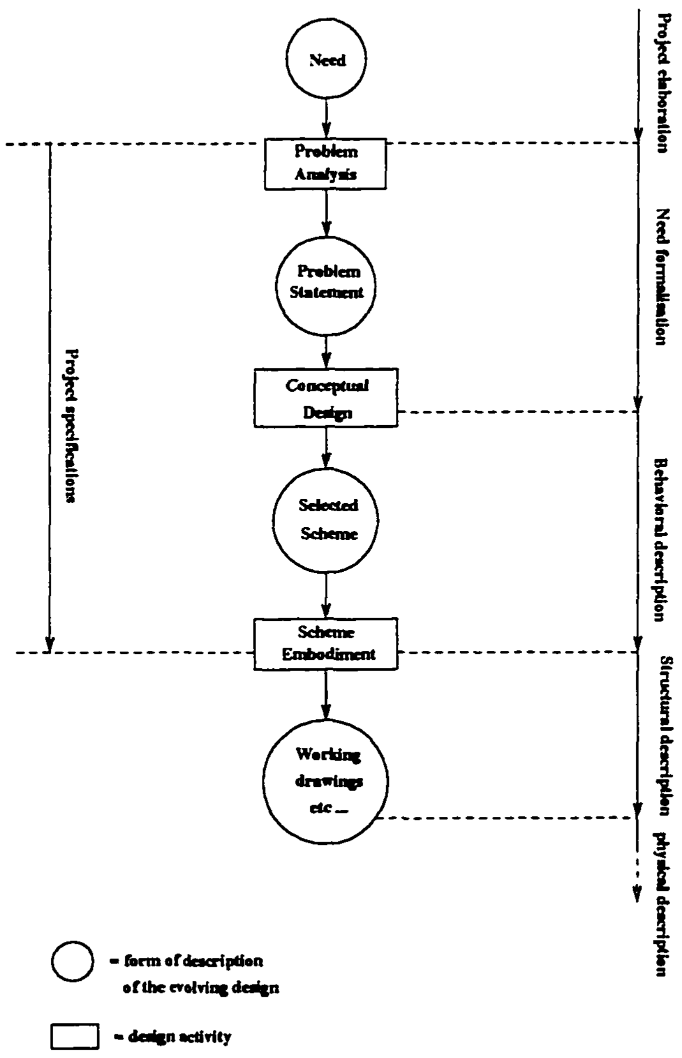


Figure 2-1: Idealistic Design Flow

A model describing design flow is shown in Fig. 2-1 (the original version was defined in [FRE85] and is described in Section 2.4) and illustrates a *top-down methodology from the statement of a need until completion.* This model is considered idealistic because no feedbacks are defined. It is assumed that each state of this design flow corresponds to the optimum design solution. Fig. 2-1 also illustrates a possible representation of the principles of first-specification and first-silicon. This idealistic model is composed of:

- *circles* which usually represent some form of description of the evolving design, although they sometimes represent a stage. For example, the circle labeled "Need" is a statement of needs which initiates the design process whereas the circle labeled "selected scheme" is a form of design description:

- *rectangles* which indicate a design activity such as analyzing the problem or performing a detailed design;

- *arrows* which sequence description forms and activities.

The first element in this design process is called "Need". When a consensus is established around a clear "statement of the problem", the "conceptual design" activity can be applied to consider different concepts (or "schemes") that can be used to solve the stated design problem. Brainstorming is required at this stage to find strategies to solve the stated problem. Thereafter, these strategies are translated into a description ("selected scheme") which then depends strongly on the requirement of the high-level attributes of the design goal, including interface constraints, size, quality, anticipated cost, and device function. The conceptual design stage is the most "open-ended" stage of the design process. The result of this conceptual design is a set of possible concepts or schemes for the design. A "scheme" is defined as an outline of major functions in the design. A scheme should be relatively explicit about special features or components but does not require much detail beyond the established practices. The next stage of the design process is called either the "embodiment of schemes" or "preliminary design". The first behavioral model is realized by implementing an initial solution. When a solution strategy is chosen, the following stages are a refining process until the final product is completed at the physical level.

This model, even though it is not feasible, illustrates clearly the top-down, first specification and first silicon concepts.

2.2 Commercial Tools

This section reviews some commercial tools which provides graphical tools to describe systems. A majority of these tools uses high-level synthesis to target programmable-logic components. Programmable-logic complexity is forcing designers into the world of HDLs and top-down design. The so-called "second wave of design engineers" are slowly moving from schematics to HDLs. In the workplace, designers are typically using a mixture of schematics, Abel-like language, and other HDLs (usually reserving HDL for a well-understood function in the design). For these design engineers, moving toward HDLis a radical change in their mind set. In order to domesticate a new design style among designers, some Electronic Design Automation (EDA) vendors (see Table 2.1 [DON96, MAN97]) provide them with graphical-entry tools facilitating the monitoring process of converting a state machine description into an HDL file.

Company	Product(s)	Types of entry accepted	HDLs generated
Alta Group of Cadence Design System	Hardware Design System	Block diagrams, state machines	VHDL, Verilog, C
Aldec	Active-CAD, Active-HDL editor	Hierarchical block diagrams, state machines, schematics	VHDL, Abel
Antares	Antares Environment Graphical Editor	Block diagrams, state machines	VHDL
Escalade	DesignBook	Block diagrams, state machines, waveforms	VHDL, Verilog
i-Logix	Express	StateCharts, activity charts, block diagram	VHDL, Verilog, C
Knowledge Base Silicon	flowHDL, blockHDL	Block diagrams, flow diagrams	VHDL, Verilog
Mentor Graphics	System Architect	State transition diagrams, state matrix, dataflow diagrams, schematics	VHDL, Verilog, C
Omniview	Alchemist	State diagrams, timing diagrams, flowcharts, truth tables	VHDL, Verilog, C
R-active Concepts	Better State Pro	StateCharts, state machines, Petri-nets	VHDL, Verilog, C, C++
Synopsis	COSSAP DSP suite, Design Source	Block diagrams	VHDL, Verilog, C

Table 2.1: Graphical HDL Code-Generation Tool Vendors

These tools are commonly classified as Electronic System Design Automation (ESDA) tools. One can notice that, from the list of vendors in Tab. 2.1, every one generates automatically VHDL code. Also, the most common type of entry accepted by these tools is the state machine. Finally, the majority of these tools can be characterized using the list of characteristics defined in Chapter 1 as follows:

$C_{ESDA} = \{$

Characteristics	Checkmark
Sequentially Decomposable Activities	✓
Concurrently Decomposable Activities	
State Transitions	✓
Immediate Mode Change	✓
Activity Completion	
Delay Specification	
Asynchronous Activities	
Design for { Testability, Manufacturing, etc }	
Multiple Model Representations	
Reusability	

$\}$

Table 2.2: Advanced High-Level Synthesis Tools Characteristics

EDA vendors claim they can ease the schematic to HDL transition with tools that generate HDLs from graphical input. For example, Aldec^{TM} has been promoting the use of state machines for programmable logic design with its Active State Editor^{TM} tool. According to Aldec^{TM}, Active State Editor^{TM} produces device-independent Complex Programmable Logic Device (CPLD) or Field Programmable Gate Array (FPGA) designs from graphical entry of bus-based state machines. In the Aldec^{TM} environment, a designer can specify combinational and sequential outputs, active clock edges, and default and trap states. The editor then converts these files into Abel and VHDL files which, according to Aldec^{TM}, are synthesis-ready.

From a designer point of view, ESDA tools are good learning tools. However, the code generated from these tools is far from being refined compared to the code written by an experienced HDL designer. Another criticism of ESDAs is that design engineers accustomed to working with schematics habitually tweak their design to correct behavioral, timing, and area problems. These designers have a hard time resisting the temptation to get into the code and fiddle with bits, even if they are not experienced HDL users. The danger here is that if the code is changed, it can disconnect from the original state-machine description. Once that happens, an error tagged during HDL simulation will not necessary connect to the original description and ESDA input is flawed.

In addition to code-generation capabilities, many ESDA tools provide a block-diagram function to help keep track of the numerous files generated during the design process. For designers used to schematics, these functions can be a useful learning tool because the top-down design methods are not just about learning code but a whole new way of thinking about a design.

The main drawback of ESDA tools, which is also true for automatic processes at every level of abstraction, is the performance of the system under design (i.e. obtaining the most efficient design at the silicon level). It is nearly impossible to get the same level of optimization with an automatic process as with a hand-written one. Then, the trade-off becomes time versus performance. Another drawback is that EDA vendors provide tools optimized for a specific architecture (FPGA, CPLD, Static Random Access Memory (SRAM) , ...). Often these vendors use benchmarks [Cor93a] provided by corporation like Programmable Electronics Performance Corporation (PREP) to promote these specialized tools. Therefore, when the targeted architecture needs to be changed it is not always a straightforward process to perform this kind of migration.

2.3 Case Study: RAM Cell

To illustrate the most advanced feature of today's CADs, we define a RAM cell. Its specification[4] is thus:

[4]This specification is used as often as possible throughout this thesis to get a common illustration of description methods and thus ease their comparison.

A list of control signals are defined: NRST (reset signal), CS (chip select), RD (read command), WD (write command) enabling identification of which action the Random Access Memory (RAM) needs to perform. An address bus allows a unique location of the data stored and manipulated using a data bus. The normal operation of the RAM is to be in a "wait state" watching for the condition " CS = '1' " to occur. When this condition is verified, the action of read or write is decoded from the combination of RD and WR (RD = '1' and WR = '0' means the RAM is in the read mode, RD = '0' and WR = '1' means the RAM is in the write mode, and other conditions than these correspond to error conditions). The RAM comes back to the wait state upon completion of its task, desired to be within 1 ns. If within this period of time the condition " NRST = '0' " is true, the RAM has to wait for the condition "NRST = '1' " to be in the wait state again. When one inconsistency on the control signals occurs, the RAM goes back to an initial state automatically after a desired time of 1 ns. When the RAM is in the initial state, a sequence of events caused by control signals (NRST = '0', NRST = '1') brings it back to the wait state.

In the Mentor Graphics^{TM} environment, a tool called System Architect^{TM} can partially capture the above specification of the RAM. First, a *context diagram* has to be created allowing the specification of the Input/Output interface as shown in Fig. 2-2. When the context diagram is defined, the functionality of the RAM needs to be described. In System Architect^{TM}, the control functions and the data transformation have to be separated. As illustrated in the data flow diagram (Fig. 2-3), the control functions are described under the node "control" and the data transformations are performed under "storage".

The control functions are described using a Moore type state machine as shown in Fig.

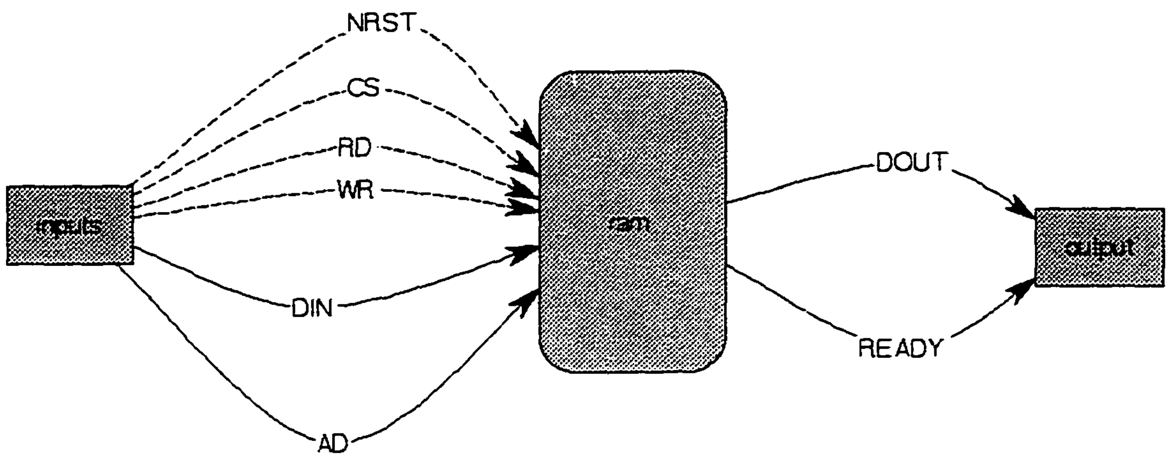


Figure 2-2: Context Diagram for RAM in the the Mentor Graphics^{TM} Design Environment

2-4. A compromise has to be made for this state machine. *In the specification, it has been defined that, for example, the RAM goes into a wait state after 1 ns when the RAM is in read or write mode. This requirement is not implemented with the description method used in this section.* Instead, this implemented duration relies on the settle time of a flip-flop component. The data transformation is described using the VHDL syntax to describe the storage function of the data. The following VHDL code is the description of the storage function input to System Architect^{TM} (the full VHDL description generated by System Architect^{TM} can be found in Appendix C):

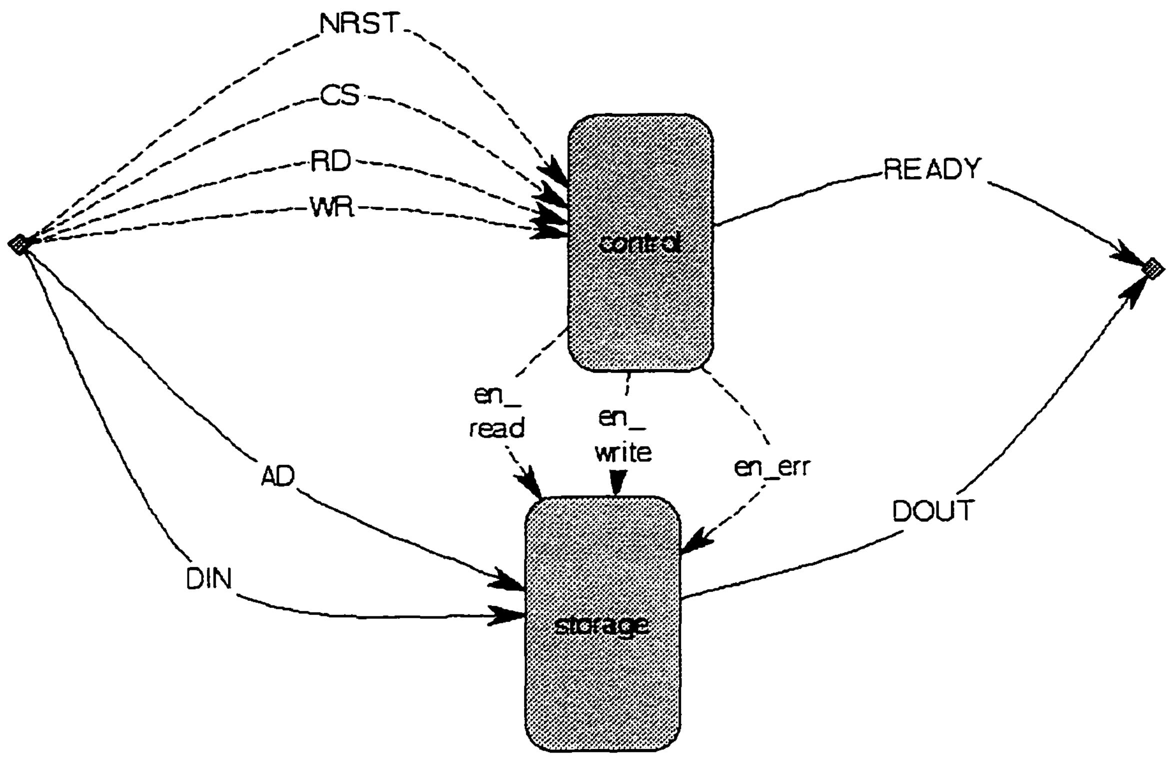


Figure 2-3: Data Flow Diagram for RAM in the Mentor Graphics^{TM}' Design Environment

```
ARCHITECTURE spec OF storage IS
BEGIN --Architecture
  -- Description of the storage activity of the RAM cell
  vhdl_storage : PROCESS (-- sensitive list of this process statement
                          AD'transaction, -- transaction is an attribute
                          DIN'transaction, -- defined in VHDL to notice
                          en_read'transaction, -- any change on a signal
                          en_write'transaction,
                          en_err'transaction)
   -- Define a list of constants : it is a nice way of programming
   CONSTANT T_READY_U : TIME := 60 ns;
   CONSTANT T_READY_D : TIME := 1 ns;
   CONSTANT T_ACCESS : TIME := 40 ns;
   CONSTANT T_WRITE : TIME := 5 ns;
   CONSTANT nb_words : INTEGER := 2 * *8;
   -- Define a new type : required in VHDL when a table of vectors needs to be used
   TYPE type_memory IS ARRAY(0 TO nb_words - 1) OF BITVECTOR(0 TO 3);
   -- Define variables : special meaning in VHDL - it is used only in a sequential
   -- statement and during simulation, the assignment of a variable is
   -- instantaneous whereas a signal has a delay
   VARIABLE prop_delay : TIME := 1ns;
```

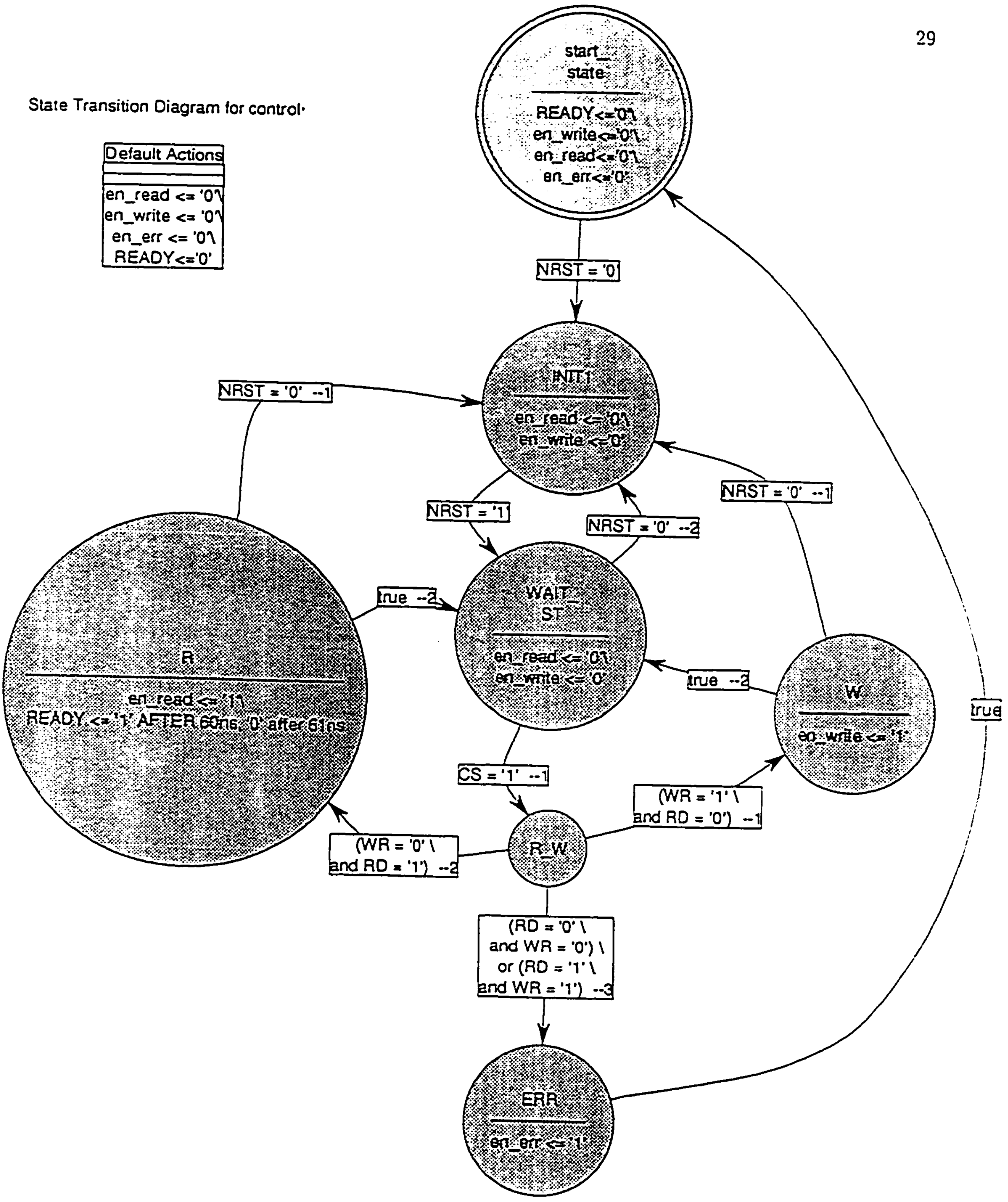


Figure 2-4: State Machine for RAM in the Mentor Graphics' Design Environment

```
VARIABLE M : type_memory;
--This function allows the conversion of a bit string to a natural number
FUNCTION value(bv : IN BITVECTOR) RETURN natural IS
VARIABLE n : NATURAL := 0;
BEGIN --process
   FOR l IN bv'low TO bv'high LOOP
        n := n * 2;
        IF bv(l) =' 1'
            THEN
                     n := n + 1;
        END IF ;;
  END LOOP;
   RETURN n;
END value; -- end of function
-- Beginning of the description of the storage function of the RAM cell
BEGIN
  IF (en_write =' 1') --the operation of writing a data in the RAM
     -- is requested
     THEN
             M(value(AD)) <= DIN AFTER T_write; -- store a data in the table M
  ELSIF (en_read =' 1') --the operation of reading a data is required
          THEN
                 DOUT <= M(value(AD)) AFTER T_access; --provide a data to the RAM
                 -- databus
  ELSIF (en_err =' 1') --an inconsistency occurs and raises an error
          THEN
                 ASSERT FALSE --statement in VHDL for simulation purposes
                 REPORT "Wrong values for WR and RD when CS rises"
                 SEVERITY WARNING;
          ELSE
                 NULL;
  END IF ;;
 END PROCESS vhdl_storage; -- end of description
END spec;
```

Notice that the above example presents a design methodology which can be characterized using the set of characteristics introduced in Chapter 1 as follows:

$C_{MG} = \{$

Characteristics	**Checkmark**
Sequentially Decomposable Activities	✓
Concurrently Decomposable Activities	
State Transitions	✓
Immediate Mode Change	✓
Activity Completion	
Delay Specification	
Asynchronous Activities	
Design for { Testability, Manufacturing, etc }	
Multiple Model Representations	
Reusability	

$\}$

Table 2.3: Mentor Graphics^{TM} Front-End Design Tool Characteristics

As indicated before, in lieu of shortcomings of the current CAD tools, the following strategy is proposed. On top of exiting CAD tools, it is desired to create a user friendly interface which would allow a seamless integration with existing CAD tools, and at the same time address the need for automation at the specification level.

Driven by the mutation of the electronics design methodologies, tools should become non-specialized description style environments for capturing high-level specifications. These environments should be graphically oriented because it is a common engineering practice to use sketches for describing the function of a system. Another important feature would be to encapsulate time without tightening the design with a clock. The method of measuring time is a design issue which must not restrict the ability to find the best solution for a system. As noted during the discussion about ESDA tools, a drawback was stated that the code generated by these tools were not optimized. Therefore, an optimization process along the same principle as the one applied to schematics needs to be defined at the specification level.

2.4 VHLLS Role

The two previous sections discussed design automation tools. Desired and logical properties for these tools is summarized as follows:

- they must provide the reduction of the design cycle time;
- they must provide an increased design quality;
- they must alleviate the design complexity of today's and tomorrow's systems;
- they must effectively maintain complex systems;
- they must facilitate improved verification facilities.

As seen in Chapter 1, these items identify the characteristics of a design automation environment. This section narrows down the design cycle to the area of interest typical for the early stage of this cycle.

The top-down methodology, as defined in Section 2.1, has been selected as an appropriate and suitable design methodology. This approach appears to be more natural and does not carry possible constraints stemming from lower levels which can reduce the spectrum of solutions, as in the case of the bottom-up approach. The objective of a top-down approach is to start with high quality specifications and inject constraints as late as possible in the process. Moreover, this approach allows for a deeper exploration of possible solutions so that problems can be solved more effectively and efficiently. Also, a top-down approach gives the opportunity to evaluate several candidate solutions before selecting the most appropriate one.

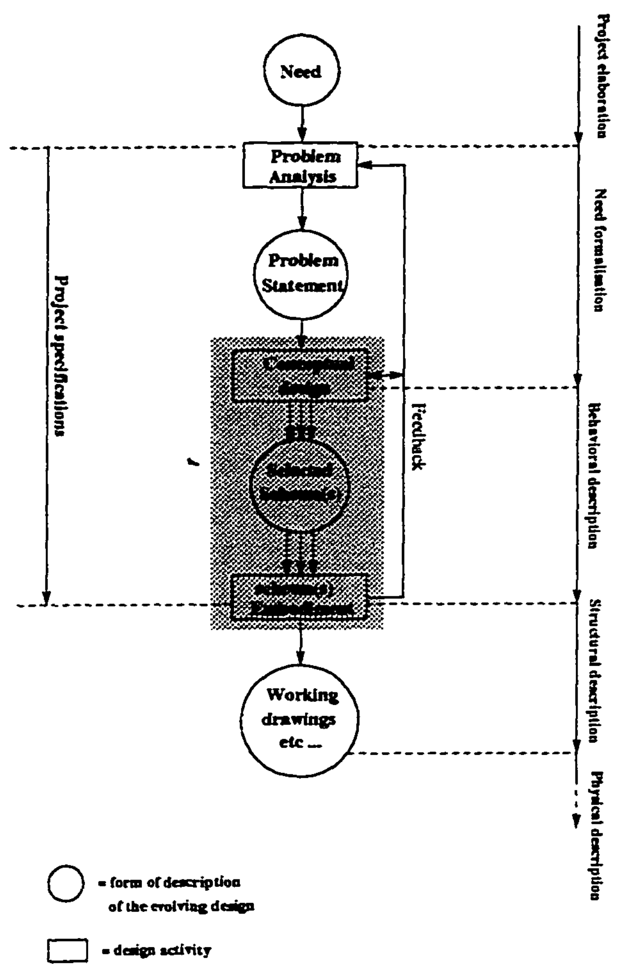


Figure 2-5: French's Design Flow

As mentioned in Chapter 1, there is a need for higher level of abstraction for design automation. For this purpose, a new level of abstraction is added as an outer ring to the Y Diagram (Fig. 1-2). This level of abstraction is called the *concept level*. The purpose of our study is to sketch out the bridge between the *concept level* and the *system level*. These two notions are explained in more depth in Chapter 3. In this context, the concept level is defined as a part of the design space along with the system level. Furthermore, these levels under study are the two highest levels. However, the concept level represents the early stage of a design flow and in that sense it precedes the system level. The corresponding model, introduced by French [FRE85], is depicted in Fig. 2-5. This model has been partially discussed in Section 2.1. Fig. 2-1 is the idealistic view of French's design flow. The main difference is in the feedback loops which characterize a refinement process of the "problem analysis" state. Indeed, during the design flow, the statement of a "need" generates an iteration of "problem statement" in order to keep the project under feasible boundaries.

The concept level can be identified by the shaded zone in Fig. 2-5. The objective here is *to develop a framework to automate the transition between a conceptual design and the embodiment of a scheme.* Indeed, the automation of these design flow sequences promotes creativity at the specification level. This automated transition is referred to as *Very High Level Logic Synthesis (VHLLS).*

Definition 2.1 ***Very High Level Logic Synthesis (VHLLS)*** *is a translation from a description at the Concept Level into a description at the System Level.*

By introducing the extra layer in the Y diagram (other additional layers are expected in the future) and defining a proper synthesis process at this new layer, we are able to address several fundamental design paradigms in a practical manner. One of them is the encapsulation of time which becomes more universal and not clock driven. The next chapter reexamines the CAD domain using a formal approach necessary for a better understanding of this matter.

Chapter 3

Very High Level Logic Synthesis (VHLLS)

The previous chapters identify the need for higher abstraction levels during the design flow. This chapter introduces the corresponding design space and a formalism associated with it. The same methodology issues are addressed using a different, more formalized approach.

3.1 Design Space Fundamentals

As indicated before, according to Thomas [TLW^{+}90] and Gajski [GK83], the design space is composed of three orthogonal domains of description:

- behavioral;
- structural;
- physical.

Fig. 3-1, commonly called the Y diagram, illustrates the three above domains. The *behavioral domain*, referred to as d_{bhv}, focuses only on the description of functions the system

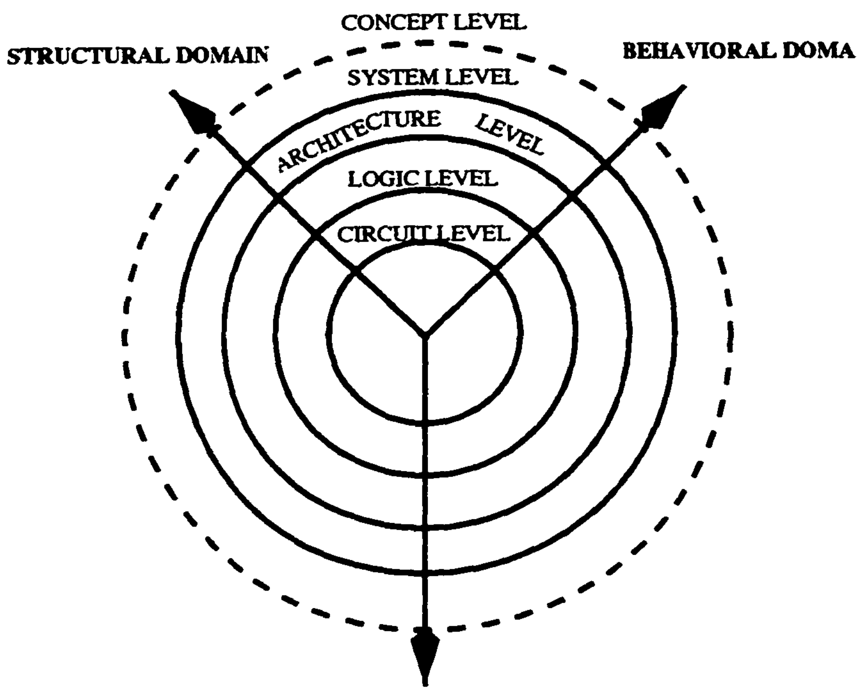


Figure 3-1: Formalization of Design Space

must perform (often referred to as the "black box" approach). In this domain, the input and output interfaces and their relationships are defined as a result. The *physical domain*, referred to as d_{phl}, focuses on the physical structure of a system under consideration. In this domain, the function of the system is not relevant. The intermediate domain which bridges the behavioral and physical domains is called the *structural domain* and is referred to as d_{str}. This domain corresponds to a mapping (or synthesis) of the behavioral domain into a set of components and connections under constraints such as cost, area, delay, etc. The system representation being in the structural domain, a second mapping process synthesizes the design into the physical domain. The origin of the orthogonal domains is the final *implementation*, referred to as ℓ_{imp}, of a system.

In these three domains, four levels of abstraction are defined:

- system level (ℓ_{sys});

- architectural level (ℓ_{act});
- logic level (ℓ_{lgc});
- circuit level (ℓ_{cct}).

The relationship between the levels of abstraction and description domains is governed by *design attributes.*

Definition 3.1 *In the design space, an* **attribute** *(a), element of the set of attributes A ($a \in A$), is either a form of representation or description by which a design is characterized. An attribute is dependent on the level of abstraction and the description domain.*

The relationships between each level of abstraction and each description domain are illustrated in Table 3.1 by their respective attributes. For example, electrical engineers are very familiar with schematics as a medium to describe the function of a system. A schematic is characterized by being in the structural domain at the logic level. The most characterizing attributes associated with this pair (structural domain, logic level) are gates, clocks, multivibrators, and flip-flops.

This section introduced the design space illustrated by the Y-diagram as shown in Fig. 3-1. The notions of level of abstractions and design domains were introduced. Each possible pair of level of abstractions and design domains is associated with a list of attributes. These attributes allow a clear distinction of each pair of level of abstractions and design domains. Note that the list of attributes is likely to evolve from a research effort on formalizing the design space. The next section proposes a formal representation of this design space as well as its associated metrics.

Abstraction Level \ Description Domain	Behavioral Domain	Structural Domain	Physical Domain
Extension ↑			
Concept Level	• Natural language description • Sketches • Mappings • Duration relationships • Math. equations	• Modules • Buses • Networks	• Boards • Boxes • Stacked MCMs
System Level	• Flowcharts • Algorithms • Regular Expressions	• Processors • Controllers • Memories • Data Pipelines • Buses	• Boards • Chips • MCMs
Architecture Level	• Register transfers	• ALUs • Multipliers • MUXs • Registers • Receivers • Transmitters • Buffers • Memories	• Chips • Floorplans • Module Floorplans • 3D-Chips
Logic Level	• Boolean equations • Waveforms • Sequencers	• Gates • Clocks • Multivibrators • Flip-Flops	• Modules • Packaging pin out • Cells
Circuit Level	• Transfer functions	• Transistors • Connections • Resistors • Capacitors • Diodes	• Transistor layouts • Wire segments • Contacts
Implementation	• Functional Documentation	• Structural Documentation	• final design

Table 3.1: Levels of Abstraction in the Design Space

3.2 Design Space Formalization

The design space is viewed as a multi-dimensional space where the directions of that space are at least the description domain and the levels of abstraction. Mathematically, the design space can be expressed as:

$$DS = < D, L, A, C, \delta, \lambda > \tag{3.1}$$

where:

- D represents the description domains in the design space DS such as $D = \{d_{bhv}, d_{str}, d_{phl}\}$
- L represents the levels of abstraction in the design space DS: $L = \{\ell_{imp}, \ell_{cct}, \ell_{lgc}, \ell_{act}, \ell_{sys}\}$. Under L, an ordered relation $<_L$ is defined as:

  $$\forall x, y \in L, x <_L y \Leftrightarrow \text{x is less abstract than y}$$

  So, the elements of L can be ordered as follows:

  $$\ell_{imp} <_L \ell_{cct} <_L \ell_{lgc} <_L \ell_{act} <_L \ell_{sys};$$

  This ordered relation (less abstract) is a relation to classify description regarding the amount of details provided to define a system. So a system description is less abstract than another (of the same system) when the information provided for describing a system is more accurate. For example, a traffic light can be described as a device to regulate traffic of terrestrial vehicles. However, a less abstract description of a traffic light is that a traffic light is a device which indicates to a driver of a vehicle either (i) to cross a junction when it is green, (ii) to stop before the junction when it is red or (iii) to be careful while crossing the junction when it is yellow.

From now on, the term *successor* of a level of abstraction x is used to mention a level of abstraction y such as $y <_L x$. For example, ℓ_{imp} is a successor of ℓ_{sys}. The term *immediate successor* of a level of abstraction x is used to mention the level of abstraction y such as $y <_L x \wedge (\nexists z \in L, y <_L z <_L x)$. For example, ℓ_{act} is the immediate successor of ℓ_{sys}. The term with the opposite meaning for successor is *predecessor*. ℓ_{sys} is a predecessor of ℓ_{cct} and ℓ_{act} is the *immediate predecessor* of ℓ_{lgc}.

- A represents a set of attributes such that $a \in A$, a being an attribute. Table 3.1 contains a non-exhaustive list of attributes. So, for example, an attribute can be ALUs, Flowcharts, or Chips;

- C represents a set of characteristics (introduced in Section 1.1.2) such that $c \in C$, c being a characteristic. For example, a characteristic can be "Sequentially Decomposable Activities";

- Mapping $\delta : D \times L \rightarrow A^* \times C^*$ associates in the design space DS a level of abstraction from L and a description domain from D onto a set of attributes from A^* and characteristics from C^*.

  A^* is the set of equivalence classes of A under $R1$ ($A^* = A/R1$). In other words, each element of A^* is an equivalence class of the elements of A under the equivalence relation $R1$. $R1$ is defined as:

$$\forall x, y \in A, x\ R1\ y \Leftrightarrow \rho(x) = \rho(y)$$

where ρ is defined as: $\rho : A \rightarrow L \times D$. So ρ defines a property that maps each attribute of A onto a description domain from D and a level of abstraction from L. For example, an element of A^* taken from the Table 3.1 is $a^* =$ {Gates, Clocks, Multivibrators, Flip-Flops} which is associated with the logic level and the structural domain. Note that each pair composed by a level of abstraction and a design domain is mapped with an element of A^* as shown in Table 3.1.

C^* is the set of equivalence classes of C under $R2$ ($C^* = C/R2$). $R2$ is defined as:

$$\forall x, y \in C, x \; R2 \; y \Leftrightarrow \zeta(x) = \zeta(y)$$

where ζ is defined as: $\zeta : C \rightarrow L \times D$. ζ defines a property that maps each characteristic of C onto a description domain from D and a level of abstraction from L. Consequently, C^* is a set of equivalence classes under the equivalence relation $R2$. For example, an element of C^* is $c^* =$ {Sequential Decomposable Activities, State Transitions, Immediate Mode Change, Activity Completion} mapped with the system level and the behavioral domain.

For example, $\delta(d_{bhv}, \ell_{sys}) =$ ({Flowcharts, Algorithm, Regular expressions}, {Sequential Decomposable Activities, State Transitions, Immediate Mode Change, Activity Completion});

- $\lambda : D \times L \rightarrow D \times L$ represents an *evolution* in the design space DS from a pair (x_1, y_1) composed by a description domain from D ($x_1 \in D$) and a level of abstraction from L ($y_1 \in L$) to another one (x_2, y_2). Just following, an interpretation of an evolution in DS is given as well as some examples.

Using the formal representation of the evolution λ in DS introduced above, commonly used evolutions on DS can be written in a mathematical form. One of these evolutions is the reverse engineering process which consists of taking an existing design description at one level of abstraction and describing it again at a higher level of abstraction. For example, if an engineer considers the description of a VLSI component at the circuit level i.e. pages of transistors ($x_1 = d_{str}, y_1 = \ell_{cct}$), the only way to understand the function of that component is to translate these pages of transistors into a description at the gate level i.e. a schematic composed of logic gates ($x_1 = d_{str}, y_1 = \ell_{lgc}$). This reverse engineering process can be applied until the engineer reaches a level of abstraction suitable for the comprehension of the component behavior. So, the evolution λ defines a reverse engineering process when:

$$\exists x, y \in L \text{ and } z \in D \text{ such that } x <_L y, \lambda(x, z) = (y, z)$$

Another common evolution on DS is the synthesis process. In general, a synthesis process is the action of combining abstract entities into a single or unified entity. In other words, a synthesis process is a process of refining a design by describing each function with a combination of less abstract functions. For example, at the system level (ℓ_{sys}), an addition between two integers i.e. $z = x + y$, where x and y are integers in $[0, 15]$ and z in $[0, 30]$, is synthesized at the logic level (ℓ_{lgc}) as follows:

$$\begin{cases} z_i = x_i \oplus y_i \oplus c_{i-1} \\ c_i = x_i y_i + x_i c_{i-1} + y_i c_{i-1} \end{cases}$$

where x_i and y_i are four bits wide bit-string (x_5 and y_5 are equal to 0), i is an index evolving from 0 to 5, z is a five bit wide bit-string, and c_i is the carry ($c_{-1} = 0$). So, the evolution

λ defined for a synthesis process is expressed formally as follows:

$$\forall x, y \in L \text{ and } w, z \in D \text{ such that } x <_L y, \lambda(y, w) = (x, z)$$

More specifically, Gajski et al. in their book [DGLW92] defines four synthesis processes as illustrated in Fig. 1-4. These synthesis processes are formalized, in a general manner, as follows:

$$\forall x \in L, \lambda(d_{bhv}, x) = (d_{str}, x)$$

because Gajski defines a synthesis process per level of abstraction as an evolution from the behavioral domain onto the structural domain. These synthesis processes are the following:

- System synthesis: $\lambda(d_{bhv}, \ell_{sys}) = (d_{str}, \ell_{sys})$ referred to as (b) in Fig. 1-4;
- Architecture synthesis: $\lambda(d_{bhv}, \ell_{act}) = (d_{str}, \ell_{act})$ referred to as (c) in Fig. 1-4;
- Logic synthesis: $\lambda(d_{bhv}, \ell_{lgc}) = (d_{str}, \ell_{lgc})$ referred to as (d) in Fig. 1-4;
- Circuit synthesis: $\lambda(d_{bhv}, \ell_{cct}) = (d_{str}, \ell_{cct})$ referred to as (e) in Fig. 1-4.

Such formalism eases the characterization of a design process in DS and gives a tool to compare design methods. In order to improve this characterization of design processes in DS, a metrical space is defined. First, the notion of distance in DS is defined allowing the introduction of a measure to evaluate a design process.

Definition 3.2 *Letting $L \times D$ be a set of paired elements. The **evolution distance** d_λ in DS is defined as a function on $(L \times D) \times (L \times D)$ into the set of non-negative real numbers. d_λ satisfies the following conditions:*

1. $\forall x, y \in D \times L, d_\lambda(x, y) = 0 \Leftrightarrow x = y$

2. $\forall x, y \in D \times L, d_\lambda(x, y) = d_\lambda(y, x)$

3. $\forall x, y, z \in D \times L, d_\lambda(x, y) \leq d_\lambda(x, z) + d_\lambda(z, y)$

The d_λ function can be written as: $d_\lambda = ||x-y||$ where $x, y \in L \times D$. Using this distance function, any discrete point in DS can be compared with any other point of DS. For example, a system synthesis process introduced above which is an evolution from the behavioral domain onto the structural domain at the system level has a evolution distance (simply referred to as distance) of 1. We write then $d_\lambda((d_{str}, \ell_{sys}), (d_{bhv}, \ell_{sys})) = ||(d_{str}, \ell_{sys}) - (d_{bhv}, \ell_{sys})|| = ||\lambda(d_{bhv}, \ell_{sys}) - (d_{bhv}, \ell_{sys})||$. Notice that if a reverse engineering evolution is performed between the structural domain and the behavioral domain at the system level, the distance of this evolution gets the same value of one.

Definition 3.3 *A* ***unary evolution*** λ_u *is defined as:*

- $\forall x \in L, \forall y, z \in D$ *with* $y \neq z$, $||(y, x) - (z, x)|| = ||\lambda_u(z, x) - (z, x)|| = 1$, *or*
- $\forall x, y \in L, \forall z \in D$ *with* $x <_L y$ *and* x *is the immediate successor of* y, $||(z, x) - (z, y)|| = ||\ \lambda_u(z, y) - (z, y)|| = 1$

A unary evolution has then the particularity of being an evolution having a distance of 1. Therefore, all the evolutions defining a synthesis process in the sense of Gajski (introduced above) are all unary evolutions. For example, logic synthesis is a unary evolution because:

$$||(d_{bhv}, \ell_{sys}) - (d_{str}, \ell_{sys})|| = 1$$

as opposed to an evolution from the behavioral domain at the system level to the structural domain at the architecture level which has an evolution distance different from 1. This distance is written then as follows:

$$||(d_{bhv}, \ell_{sys}) - (d_{str}, \ell_{act})|| \neq 1$$

Having defined a unary evolution, an evolution can then be viewed as a sequence of unary evolutions which brings the design from one point in DS to the desired one.

Lemma 3.1 *If an evolution is not unary, then there may be a* ***composition*** *of unary evolutions such as:*

$$\lambda(x, y) = \lambda_u \circ \ldots \circ \lambda_u(x, y)$$

where $||(x_{i+1}, y_{i+1}) - (x_i, y_i)|| = ||\lambda_u(x_i, y_i) - (x_i, y_i)|| = 1$

Proof:

If $\lambda_0(x_0, y_0) = (x_1, y_1)$ such that $d_\lambda\ ((x_0, y_0), (x_1, y_1)) = 1$ then $\lambda_0(x_0, y_0) = \lambda_u(x_0, y_0)$ is true.

Let us assume that

$$\lambda_n(x_0, y_0) = (x_n, y_n) = \underbrace{\lambda_u \circ \ldots \circ \lambda_u}_{n}(x_0, y_0)$$

is true with $d_\lambda\ ((x_i, y_i), (x_{i+1}, y_{i+1}) = 1$.

The evolution $\lambda_{n+1}(x_0, y_0) = (x_{n+1}, y_{n+1})$ can be written as a sequence of evolutions such as $\lambda_n(x_0, y_0) = (x_n, y_n)$ and $\lambda(x_n, y_n) = (x_{n+1}, y_{n+1})$. The distance of the last evolution is 1 and then $\lambda_u(x_n, y_n) = (x_{n+1}, y_{n+1})$. So we get that

$$\lambda_{n+1}(x_0, y_0) = \lambda_u \circ \lambda_n(x_n, y_n) = (x_{n+1}, y_{n+1}).$$

Because

$$\lambda_n(x_0, y_0) = (x_n, y_n) = \underbrace{\lambda_u \circ \ldots \circ \lambda_u}_{n}(x_0, y_0),$$

we can rewrite $\lambda_{n+1}(x_0, y_0) = (x_{n+1}, y_{n+1})$ as

$$\lambda_{n+1}(x_0, y_0) = \lambda_u \circ \underbrace{\lambda_u \circ \ldots \circ \lambda_u}_{n}(x_0, y_0) = \underbrace{\lambda_u \circ \lambda_u \circ \ldots \circ \lambda_u}_{n+1}(x_0, y_0)$$

which proves that an evolution is a composition of unary evolutions.

□

Using this result, the notion of distance can be improved by saying that a distance of an evolution is the sum of the distances of each unary evolution which composes this evolution.

Proposition 3.1 *The distance of an evolution is defined as:*

$$\forall (x, y) \in D \times L,\ d_\lambda = \|\lambda(x, y) - (x, y)\| = \sum_i \|\lambda_u(x_i, y_i) - (x_i, y_i)\|$$

where $\|\lambda_u(x_i, y_i) - (x_i, y_i)\| = 1$

As an illustration of the above notions, let us consider that the desired evolution in DS is the following:

$$\lambda(d_{bhv}, \ell_{act}) = (d_{str}, \ell_{lgc}),$$

which is the evolution of today's commercial synthesis tools. One possible composition is:

$$\lambda_u(d_{bhv}, \ell_{act}) = (d_{str}, \ell_{act})$$

$$\lambda_u(d_{str}, \ell_{act}) = (d_{str}, \ell_{lgc})$$

Using this composition, its distance is then equal to:

$$\|(d_{bhv}, \ell_{act}) - (d_{str}, \ell_{lgc})\| = 2$$

This example shows that when an electrical engineer designs a system at the architecture level using VHDL for synthesis and then applies a synthesis process to this design in order to get a schematic, implicitly, he or she uses indeed two evolutions.

Notice that the decomposition of an evolution is not unique. Also, the feasibility of all evolutions is not guaranteed with today's tools. Further research needs to be performed to characterize all the possible evolution decompositions. For example, the evolution from the physical domain to either the behavioral or structural domain at the system, architecture, and logical levels has not been performed so far.

In this section, a well-established design space, known as the Y-diagram (see Fig. 3-1) was presented. A formal description of it was proposed allowing a formal definition of design processes such as synthesis or reverse engineering processes. Moreover, the notion of evolution in the design space was defined as well as a formal characterization of the level of abstractions and design domains. Along with this formalization, some metrics which enable another method of comparison between design processes was defined.

3.3 Extended Design Space

Currently, toolmakers provide efficient CAD applications which perform Register Transfer (RT) synthesis. However, there is still a missing link between conceptual specification and system level description in the behavioral domain. In Fig. 3-1, an extra layer, called the *Concept Level*, has been added to the original Y diagram. This extra layer provides a framework for the introduction of the next generation of CAD tools which will be more characterized thanks to the formalism defined in the previous section.

Definition 3.4 *The **concept level**, referred to as ℓ_{cpt}, is a level of abstraction characterized in each description domain by attributes more abstract than those at the system level.*

A classification of these attributes into the three description domains is shown in Table 3.1. The list of attributes in each domain is not exhaustive. For example, the attributes characterizing the concept level in the behavioral domain are Natural Language Descriptions, Sketches, Mappings, Duration Relationships, etc. In other words, at the concept level, the behavior of a system are specified using a description methods such as natural language, sketches, and so on.

The addition of the concept level in L extends the design space DS. So, the design space DS becomes the *Extended Design Space (EDS)*. Mathematically, the Extended Design Space can be expressed as:

$$EDS = < D, L_{ext}, A_{ext}, C_{ext}, \delta_{ext}, \lambda_{ext} > \tag{3.2}$$

where:

- D represents the description domains in the extended design space EDS: $D = \{d_{bhv}, d_{str}, d_{phl}\}$;
- L_{ext} represents the levels of abstraction in the extended design space EDS: $L_{ext} = \mathrm{L} \cup \{\ell_{cpt}\}$ such that ℓ_{cpt} is the immediate predecessor of ℓ_{sys} ($\ell_{sys} <_L \ell_{cpt}$):
- A_{ext} represents a set of attributes such that $A \subset A_{ext}$ and the additional elements of A_{ext} are attributes derived for L_{ext}. Therefore, the extra attributes, as illustrated in Table 3.1, are natural language, sketches, modules, buses, boards, boxes and so on;
- C_{ext} represents a set of characteristics such that $C \subset C_{ext}$ and the additional elements of C_{ext} are characteristics derived for L_{ext}. These extra characteristics, as introduced in Chapter 1, are among others delay specification, multiple model representations and so on;

- $\delta_{ext} : D \times L_{ext} \rightarrow A^*_{ext} \times C^*_{ext}$ associates in the extended design space EDS a level of abstraction from L_{ext} and a description domain from D to a set of attributes from A^*_{ext} and characteristics from C^*_{ext}.

  A^*_{ext} is the set of equivalence classes of A_{ext} under $R1_{ext}$ ($A^*_{ext} = A_{ext}/\ R1_{ext}$). In other words, each element of A^*_{ext} is an equivalence class of the elements of A_{ext} under the equivalence relation $R1_{ext}$. $R1_{ext}$ is defined as:

$$\forall x, y \in A_{ext}, x\ R1_{ext}\ y \Leftrightarrow \rho_{ext}(x) = \rho_{ext}(y)$$

  where ρ_{ext} is defined as: $\rho_{ext} : A_{ext} \rightarrow L_{ext} \times D$. So ρ_{ext} defines a property that maps each attribute of A_{ext} onto a description domain from D and a level of abstraction from L_{ext}. For example, an element of A^*_{ext} taken from the Table 3.1 is a^*_{ext} = {Modules, Buses, Networks} which is associated with the concept level and the structural domain.

  C^*_{ext} is the set of equivalence classes of C_{ext} under $R2_{ext}$ ($C^*_{ext} = C_{ext}/\ R2_{ext}$). $R2_{ext}$ is defined as:

$$\forall x, y \in C_{ext}, x\ R2_{ext}\ y \Leftrightarrow \zeta_{ext}(x) = \zeta_{ext}(y)$$

  where ζ_{ext} is defined as: $\zeta_{ext} : C_{ext} \rightarrow L_{ext} \times D$. ζ_{ext} defines a property that maps each characteristic of C_{ext} onto a description domain from D and a level of abstraction from L_{ext}. Consequently, C^*_{ext} is a set of equivalence classes under the equivalence relation $R2_{ext}$. For example, an element of C^*_{ext} is c^*_{ext} = {Sequential Decomposable Activities, State Transitions, Immediate Mode Change, Activity Completion, Concurrently Decomposable Activities, Asynchronous Activities, Multi Model Representation, Reusability, Design for {Testability, Manufacturing, ...}} mapped with the concept level and the behavioral domain.

As a result, we can state that $\delta_{ext}(d_{bhv},\ \ell_{cpt}) = (\{$Natural language description, Sketches, Mappings, Duration relationships, Math. equations$\}$, $\{$Sequential Decomposable Activities, State Transitions, Immediate Mode Change, Activity Completion, Concurrently Decomposable Activities, Asynchronous Activities, Multi Model Representation, Reusability, Design for $\{$ Testability, Manufacturing, $\ldots\}\})$;

- $\lambda_{ext} : D \times L_{ext} \rightarrow D \times L_{ext}$ represents an evolution process in the extended design space EDS from a pair consisting of a description domain from D and a level of abstraction from L_{ext} to another one. Just following, an interpretation of an evolution in EDS is given as well as some examples.

Note that, to simplify the notations from now on, the index *ext* is dropped from the above notation. Using the extended design space formalism, two new evolution processes can be introduced: *Concept synthesis* and *Concept refinement.* The composition of these two evolutions defines VHLLS as shown in Fig. 1-4. In a sense of synthesis defined by [DGLW92], the concept synthesis is defined as follows:

Definition 3.5 *The **concept synthesis** is an evolution from the behavioral to the structural domain such that:*

$$\lambda_{ext}(d_{bhv}, \ell_{cpt}) = \lambda_{cpt-synt}(d_{bhv}, \ell_{cpt}) = (d_{str}, \ell_{cpt})$$

The concept synthesis is on the top of other synthesis processes introduced in Section 3.2. Concept synthesis starts with a set of general information about a desired behavior through shared variables or message passing. It generates a structure of modules and networks. Each module can be described by a behavioral description at the system level. This refinement process is performed through an evolution referred to as the concept refinement. This evolution is defined as follows:

Definition 3.6 *The* ***concept refinement*** *is an evolution allowing the refinement of a system description from the concept level into the behavioral level such that:*

$$\lambda_{ext}(d_{str}, \ell_{cpt}) = \lambda_{cpt-reft}(d_{str}, \ell_{cpt}) = (d_{bhv}, \ell_{sys})$$

For example, if we specify in a natural language the behavior of a traffic light by saying "A traffic light system regulates the flow of terrestrial vehicles at a junction of two bidirectional roads". Applying a concept synthesis of this specification results in networks with four modules. Each module represents a traffic light. The concept refinement is considering each module and providing it a behavioral description at the system level such as a flowchart specifying how the traffic light can change color.

Section 2.4 of the previous chapter describes the VHLLS process and provides a general definition of VHLLS. With the introduction of the above formalism to describe an evolution in the design space, we can provide a more formal definition of VHLLS.

Definition 3.7 ***Very High Level Logic Synthesis (VHLLS)*** *is a composition of two evolutions: concept synthesis and concept refinement such as:*

$$\lambda_{ext}(d_{bhv}, \ell_{cpt}) = \lambda_{cpt-reft} \circ \lambda_{cpt-synt}(d_{bhv}, \ell_{cpt}) = (d_{bhv}, \ell_{sys})$$

The definition of VHLLS leads toward the definition of a new generation of CAD tools which can be characterized using the measurement schema for classification and comparison of design methodologies defined in this chapter. Chapter 4 illustrates that no commercial tools meet the characterization of CAD tools able to perform the VHLLS methodology.

The next chapter seeks out a process to perform VHLLS. Methods under research are reviewed, classified and analyzed. As a result, the definition of a new synthesis process more suitable for VHLLS is introduced in Chapter 5.

Chapter 4

Formal Mechanisms for VHLLS

In the previous chapters, the rationale for a VHLLS process is stated and formalized. As the next logical step, the implementation of a such process needs to be considered. To this end, formal mechanisms suitable to perform the VHLLS process are prescribed for consideration in this chapter. In the scope of this research, two families of description models have been selected and introduced in Section 4.1. A separate section is entirely dedicated to each of these two families detailing the most relevant description models in each. As a concluding part of these two sections, a comparison of the presented methods is performed. Their advantages and disadvantages regarding their impact on the characterization of VH-LLS (referred to as C in Chapter 1) are highlighted. These methods are then compared to the minimum characterization set (referred to as C_{min} in Chapter 1. C_{min} leads to a new generation tool called Specification Procedure for Electronic Circuits in Automation Language (SPECIAL)).

4.1 Taxonomy

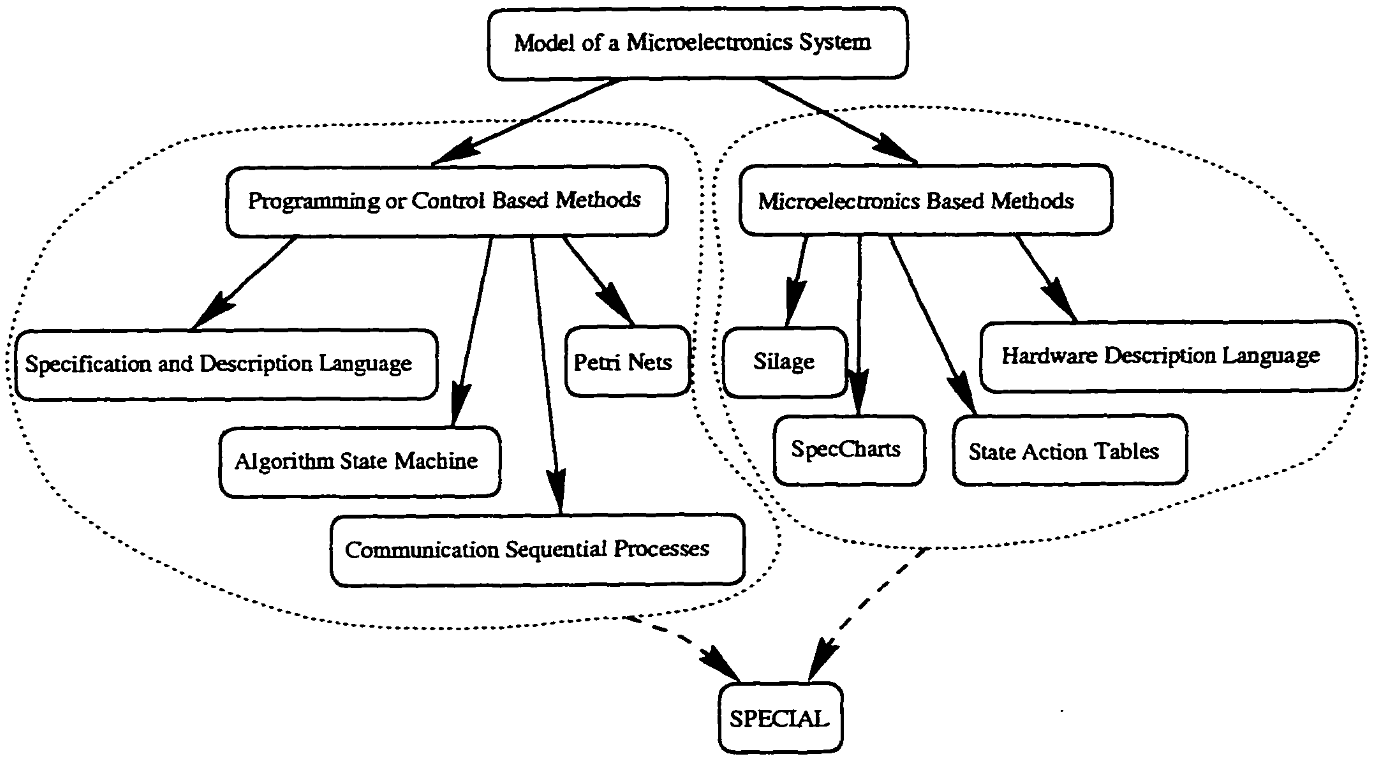

Model of a Microelectronics System
Programming or Control Based Methods
Microelectronics Based Methods
Specification and Description Language
Petri Nets
Silage
Hardware Description Language
Algorithm State Machine
SpecCharts
State Action Tables
Communication Sequential Processes
SPECIAL

Figure 4-1: Taxonomy of Formal VHLLS Mechanisms

We consider the two most appropriate families of methods to specify and describe a microelectronics system as shown in Fig. 4-1:

- programming or control based methods (described in Section 4.2);
- microelectronics based methods (described in Section 4.3).

The first family of description methods (programming or control based methods) has taken its heritage from both the computer and automatic control process areas. For instance, the Petri nets method is a typical approach of solving control problems. It is used to describe distributed systems with emphasis on concurrent, non-deterministic processes and on problems of communication and synchronization. In the computer area, the communication sequential processes method has been developed to overcome the limitations of the traditional programming languages with respect to programs running on a multi-processor machine. The choice of investigating these methods is relevant because, at the concept level, many similarities appear between the two families: sequentiality, concurrence of processes, etc. This family (programming or control based methods) is important to investigate because the approach of tackling a design problem is culturally different compared with the microelectronics world. For instance, control methods decompose a problem more easily into concurrent sub-problems compared to a microelectronics problem which is decomposed into sequential sub-problems.

The second family of methods (microelectronics based methods) has taken its heritage from the microelectronics area. The most typical methods are the Hardware Description Languages (HDLs) and in particular two of the most popular ones: VHDL and Verilog^{TM}. These two description methods use the principle of programming language to describe hardware behavior as presented in Section 4.3.1.

As defined in Chapter 1, the next generation of CAD must include the functionalities not only of today's commercial tools, but also additional features provided in the list of characteristics C. As indicated before, the tool to implement the next generation of CAD tools which meets the requirements of C_{min} is called Specification Procedure for Electronic Circuits in Automation Language (SPECIAL). SPECIAL must have the ability to offer to designers the most suitable description methods from, at least, the two families under study.

Note that the next two sections present description methods from the two description families introduced above. These sections are self-contained with respect to the notation and symbols. The description of each of these methods is very brief. If the reader wishes to probe these methods further, references are provided. Also, each description method is illustrated using the case study introduced in Chapter 2. Comparisons of and remarks about these methods are made in the last sub-section of each family section.

4.2 Process Control Based Methods

This section presents description methods which originate in the automatic control theory and algorithmic fields.

4.2.1 Algorithmic State Machines

Introduced by Clare, the Algorithmic State Machine (ASM) chart [CLA73] is a diagrammatic description[1] of the output function and the next-state of a FSM. It resembles a conventional flow chart where a control flow is expressed graphically while an operational behavior is described using textual assignment statements. So, ASM can be viewed as a super-set of FSM.

Three basic graphical components allow a construction of ASM charts:

- *state box*: contains a list of either register operations or output signal names that the controller generates while in this state. The exit path of the state box leads to other state boxes, decision boxes or conditional output boxes. The exit path is represented by a rectangle;
- *decision box*: describes the effect of an input on the controller. Two exit paths can be taken regarding the enclosed condition: one when that condition is true, the other when it is false. The shape of the decision box is in a diamond;

[1]A description in the form of diagram like an algorithmic chart

- *conditional box:* describes register assignments or outputs which are dependent on one or more inputs in addition to the state of the FSM. The rounded corners of a conditional box differentiate it from the state box.

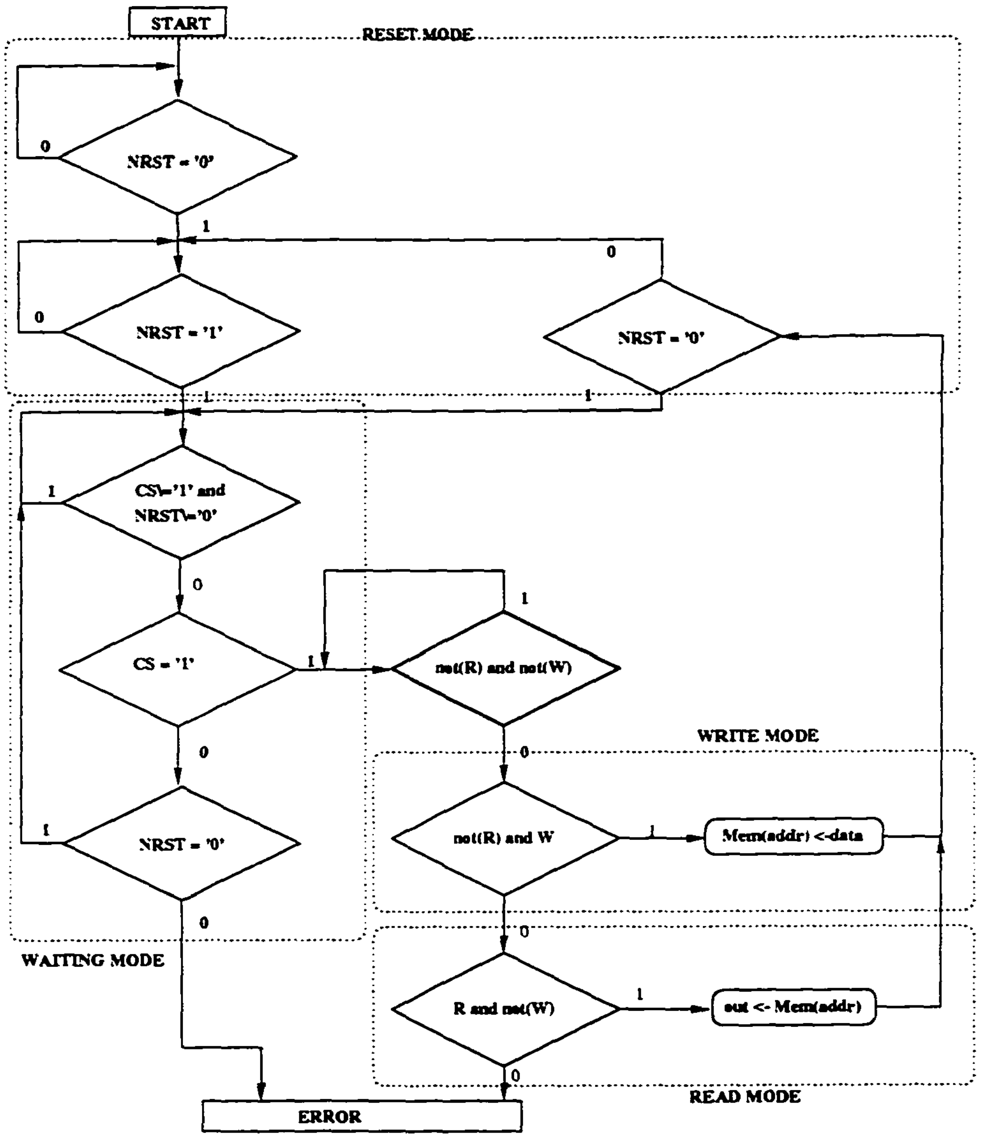


Figure 4-2: RAM Description in ASM

Another structure, called block, is defined in the ASM chart. A block consists of one state box and the decision and conditional boxes connected to its exit path. One characteristic of a block is that it has one entrance and any number of exit paths represented by the structure of the decision boxes. One block describes the FSM operation during one state. So, an ASM chart is a interconnection of blocks. Like the FSM, the timing model is a one-phase synchronous clocking scheme. Therefore, delay specifications in ASM charts, as introduced in Chapter 1, are not possible due to its dependence on a global clock.

As an example, we use the specifications of the RAM cell introduced in the case study in Section 2.3. The ASM chart can easily represent the sequence of events which initializes the RAM cell as shown in Fig. 4-2 from the "START" box until the diamond box labelled " CS = '1' ". This sequence is described using decision boxes. So, starting from the state box "START", a reset sequence is applied using two decision boxes conditioned by the value of the reset signal. After the reset sequence, the RAM cell goes to a wait mode for a chip select signal to occur. The wait mode is modeled by a perpetual scanning operation of the signal CS. So, when the chip select occurs, the RAM cell is either in a read mode or write mode. In both modes, a conditional box is used either to modify the outputs of the cell (read from the RAM) or to apply a storage operation (write into the RAM). Finally, the RAM cell returns into the wait mode unless a reset is required.

4.2.2 Communication Sequential Processes

The Communication Sequential Process (CSP) [HOA78] language was developed to overcome the limitations of the traditional programming languages with respect to programs running on multi-processor machines. This language follows the basic idea that systems can be decomposed into subsystems which operate concurrently and interact with each other as well as with their common environment.

A CSP program consists of *processes P* which stand for the behavior pattern of an object

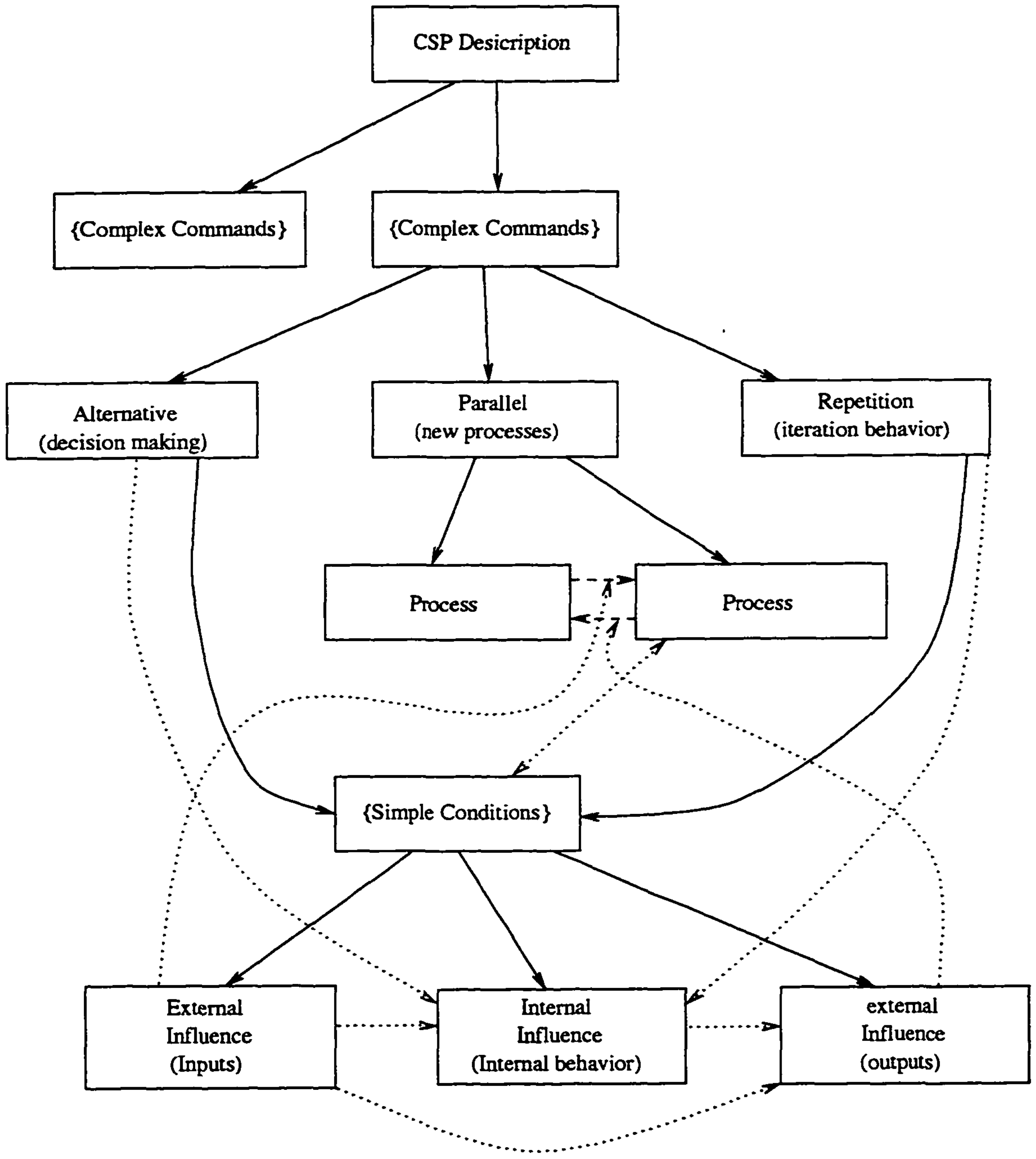

CSP Desicription
{Complex Commands}
{Complex Commands}
Alternative (decision making)
Parallel (new processes)
Repetition (iteration behavior)
Process
Process
{Simple Conditions}
External Influence (Inputs)
Internal Influence (Internal behavior)
external Influence (outputs)

Figure 4-3: CSP Hierarchical Structure and Interactions

as well as its environment and the system described by all the objects. So, within a system, processes act and interact with each other as they evolve concurrently as illustrated in Fig. 4-3 (Note that a plain arrow represents an inheritance from the starting box and a dashed arrow shows the interaction between the elements of the graph).

Hence, a system is described using a list of command (represented in Fig. 4-3 by round brackets surrounding "Complex commands" or "Simple commands" i.e. { Complex commands }) describing processes and their interaction with each other. A command specifies the behavior of a device executing the command. The command list specifies a sequential execution ordering of the commands in the list. There are two classes of commands. The first class refers to simple commands which contribute to altering the internal state of the executing process, affecting the external environment, and affecting both the internal state as well the external environment as in the input command. The second class refers to complex commands which are structured commands and involve the execution of all their constituent commands. This last class of commands contains the structure for decision making, parallel behavior of processes and implementation of interactive behavior.

In CSP some processes are created to encompass a control construction. So, if event x and process P are involved in constructing a command, $(x \rightarrow P)$ describes an object which first engages in event x (meaning when x occurs) and then behaves exactly as described by P. That can be referred to as *guarded commands*. Such a structure can lead to *non-deterministic*[2] behavior which is a salient difference between CSP and most other languages.

Communication between concurrent processes is simply specified with explicit *input* and *output* commands. That is possible only under three main conditions: (i) the output

[2]The description of a system may lead to a case where the decision making mechanism does not generate a unique process activation.

command in one process specifies another process as the destination of the data to be sent; (ii) the input command of a process using data from other processes needs to include the source of the data to be received; and (iii) match of data type during communications between processes.

To illustrate the CSP description style, we use the case study introduced in Section 2.3. So, the description of the RAM cell becomes a sequence of statements. The CSP description is as follows:

- List of events denoted as $a, b, c, d, e, f, g, out.ready, out.dout, in.ad, in.din$ where
  - a = " NRST = '0' " meaning the reset command is active;
  - b = " NRST = '1' " meaning the reset command is not active;
  - c = " CS = '1' " meaning the RAM cell is selected;
  - d = " CS = '0' " meaning the RAM cell is not selected;
  - e = " (WR ='0') and (RD = '1') " meaning the RAM is in read mode;
  - f = " (WR ='1') and (RD = '0') " meaning the RAM is in write mode;
  - g = " ((WR ='0') and (RD = '0')) or ((WR ='1') and (RD = '1')) " meaning the RAM is in error mode;
  - *out.ready* = "output a ready pulse" meaning the RAM cell is ready to send data out;
  - *out.dout* = "output dout" meaning the selected value is sent;
  - *in.ad* = "input ad" meaning the address of the data to provide or store is given;
  - *in.din* = "input din" meaning a data is provided to store;
  - *out.error* = "output error message" meaning an error occurs;

- Definition of the processes:
  - $WAIT$ meaning that the RAM cell is in wait mode waiting for a read or write operation;
  - RW meaning that the RAM cell is selected and needs to identify its mode of operation;
  - $READ$ meaning that the RAM cell is in read mode;
  - $WRITE$ meaning that the RAM cell is in write mode;
  - ERR meaning that the RAM cell is in error mode;
- Specification of the RAM cell:
  - $\alpha RAM = \{a, b\}$ meaning the alphabet of RAM is a and b (list of events involved in the description of the process RAM);
  - $RAM = (a \rightarrow b \rightarrow WAIT)$ meaning that initially, the RAM cell needs to acknowledge the event a following by the event b before applying process WAIT;
  - $\alpha WAIT = \{a, b, c\}$ meaning the alphabet of $WAIT$ is a, b and c (list of events involved in the description of the process $WAIT$);
  - $WAIT = \mu WAIT.(a \rightarrow b \rightarrow WAIT$

    $\qquad\qquad | \; c \rightarrow RW)$

    meaning that either a reset sequence occurs or the RAM is selected;
  - $\alpha RW = \{e, f, g\}$ meaning the alphabet of RW is e, f and g (list of events involved in the description of the process RW);
  - $RW = (e \rightarrow READ$

    $\qquad | \; f \rightarrow WRITE$

$| g \rightarrow ERR)$

meaning that the RAM cell is either in a read, write or error mode;

- $\alpha READ = \{out.ready, in.ad, out.dout\}$ meaning the alphabet of $READ$ is $out.ready$, $in.ad$ and $out.dout$ (list of events involved in the description of the process $READ$);
- $READ = (out.ready \rightarrow in.ad \rightarrow out.dout \rightarrow WAIT)$

  meaning that the RAM cell is sending the data selected by the provided address;
- $\alpha WRITE = \{out.ready, in.ad, in.din\}$ meaning the alphabet of $WRITE$ is $out.ready$, $in.ad$ and $in.din$ (list of events involved in the description of the process $WRITE$);
- $WRITE = (out.ready \rightarrow in.ad \rightarrow in.din \rightarrow WAIT)$

  meaning that the RAM cell is receiving data to store at the provided address;
- $\alpha ERR = \{out.error, a, b\}$ meaning the alphabet of ERR is $out.error$, a and b (list of events involved in the description of the process ERR);
- $ERR = (out.error \rightarrow a \rightarrow b \rightarrow WAIT)$

  meaning that the RAM cell does not recognize the event sequence;

The description using CSP does not entirely follow the specification for RAM cell given in Section 2.3 because the specifications say that when the RAM cell is either in read or write mode, it has to wait 1 ns before going to the WAIT state unless a reset command occurs. For the above description, we overrule this specification issue by noticing that by going in a wait state immediately, if a reset command is active, the RAM performs the reset cycle as well.

4.2.3 Petri Nets

Petri nets [REI85] were introduced by Petri in the early 1960s as a mathematical tool for modeling distributed systems and, in particular, notions of concurrency, non-determinism, communication and synchronization. There are many varieties of Petri nets from black and white nets, which are conceptually simple and straightforward to analyze, to more complex nets such as colored nets which allow the modeling of complex systems. A simple (black and white) Petri net is a bi-partite graph with nodes which may be places (drawn as circles) or transitions (drawn as rectangles or lines). Edges can connect places to transitions (known as input arcs, with the corresponding places known as input places) or transitions to places (known as output arcs, and the corresponding places known as output places). A Petri net can be marked by indicating tokens which are contained in each place at a point in time (drawn as dots). If all the input places of a transition contain (at least) one token, then the transition is eligible for firing. If it does fire then one token is removed from each of its input places and one token is added to each of its output places. A Petri net is executed by establishing an initial marking and then, at each subsequent cycle, choosing a set of eligible transitions for firing. Notice that the ability of a transition to fire is determined solely by local conditions, namely the presence of tokens in the adjacent input places. This locality of reference is a desirable feature in modeling concurrent systems. Even with the simplicity of black and white nets, it is possible to model interesting concurrent systems.

Petri nets have already been used to specify the behavior of a system [PB91]. Such systems are synchronous parallel controllers. Having their specifications defined with Petri nets, a synthesis process can be applied targeting a VHDL description at the RT level. In the formalism defined in Chapter 3, the above synthesis process can be described as a composition of a system synthesis process and a system refinement or more formally:

$$\lambda_{syst-synth}(d_{bhv}, \ell_{sys}) = (d_{str}, \ell_{sys})$$

$$\lambda_{syst-reft}(d_{str}, \ell_{sys}) = (d_{bhv}, \ell_{act})$$

$$\text{then } \lambda_{Petri-net-synth}(d_{bhv}, \ell_{sys}) = \lambda_{syst-reft} \circ \lambda_{syst-synth}(d_{bhv}, \ell_{sys}) = (d_{bhv}, \ell_{act})$$

A VHDL template has been defined to meet criteria such as a direct match with the Petri net schematics and also to be compatible with simulator and synthesis packages. The VHDL code generated is into the VHDL synthesis's subset.

As an illustration of Petri nets (Fig. 4-4), the RAM cell specifications introduced for the case study in Section 2.3 is used. The Petri net graph is a dynamic graph meaning that a token is moving from place P#i to place P#j e.g. in Fig. 4-4, initially one token in present at place P#1 and can move to place P#2. With the token in P#1, the transition T#1 authorize the token to move to P#2 when the condition associated with T#1 is verified. In the case of T#8, the transition occurs in any circumstances but depending on the evaluation of the condition associated with the transition the token can go either in P#3 (when "NRST = '0"' is true) or in P#2 (when "NRST = '0"' is false). When the token is in one place, actions can be executed. In particular, when the token is either in P#5 then the action of reading in the memory is activated or in P#6 then the action of writing in the memory is activated or in P#7 then an error alarm occurs. In the RAM example, each transition T#i has a condition associated with it as follows:

- T#1 being: NRST= '0';
- T#2 being: NRST= '1';

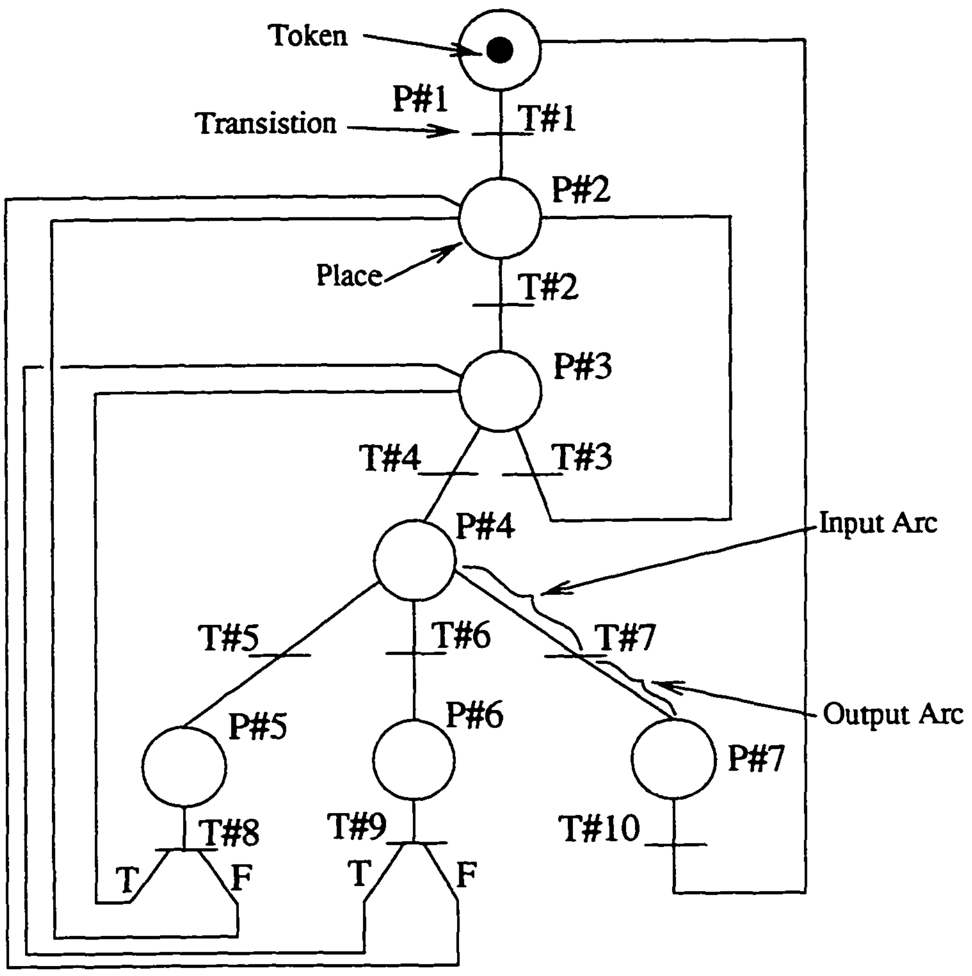


Figure 4-4: RAM Using Petri nets

- T#3 being: NRST= '0';
- T#4 being: CS = '1';
- T#5 being: RD= '1' AND WR= '0';
- T#6 being: RD= '0' AND WR= '1';
- T#7 being: (RD= '0' AND WR= '0') OR (RD= '1' AND WR= '1');
- T#8 being: NRST= '0';
- T#9 being: NRST= '0';
- T#10 being: True.

Once again, the specifications given for the RAM cell are not fully implemented. Using Petri nets, no time delay can be specified.

4.2.4 Specification and Description Language

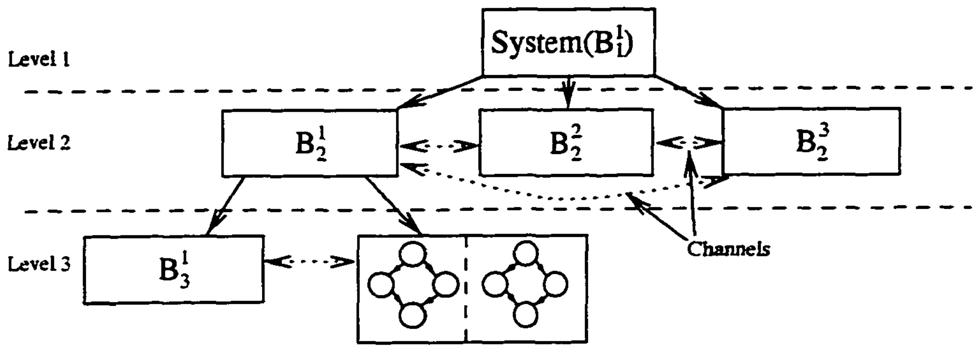


Figure 4-5: SDL Hierarchical Structure

Specification and Description Language (SDL) [BS91] is a language for the specification and description of systems. This language, mainly used in the telecommunication field, is well suited for specifying real-time and interactive systems. In a nutshell, SDL essentially specifies the behavior of the system and its interaction with its environment.

The basis for describing a system behavior (shown in Fig. 4-5) is a hierarchy (tree like) of dataflow diagrams (to create the "branches" of the tree) and state machine at the leaf level. The element of the tree referred to as *block* represents the main structuring concept in SDL. A block helps partition a system description into sub-descriptions. So, block B_i^n can be composed by interrelated sub-blocks B_{i+1}^p. We say B_i^n is the n-th block at the level i and B_{i+1}^p is the p-th block of the level $i+1$ a level under level i (e.g. in Fig. 4-5), we have three levels. Level 1 is the root level which represents the system description. A partition of level 1 is composed by three blocks at level 2 (a sub-level of level 1). Level 3, in Fig. 4-5, represents the partition of only one block B_2^1 into one block and one leaf. A leaf in the SDL tree as shown in Fig. 4-5 has one or more *processes*. A process is essentially a state machine which works concurrently with other processes. A mechanism for exchanging information between blocks of the same level is modeled with arrow type of links called *channels*. In Fig. 4-5, channels are represented with dashed arrows.

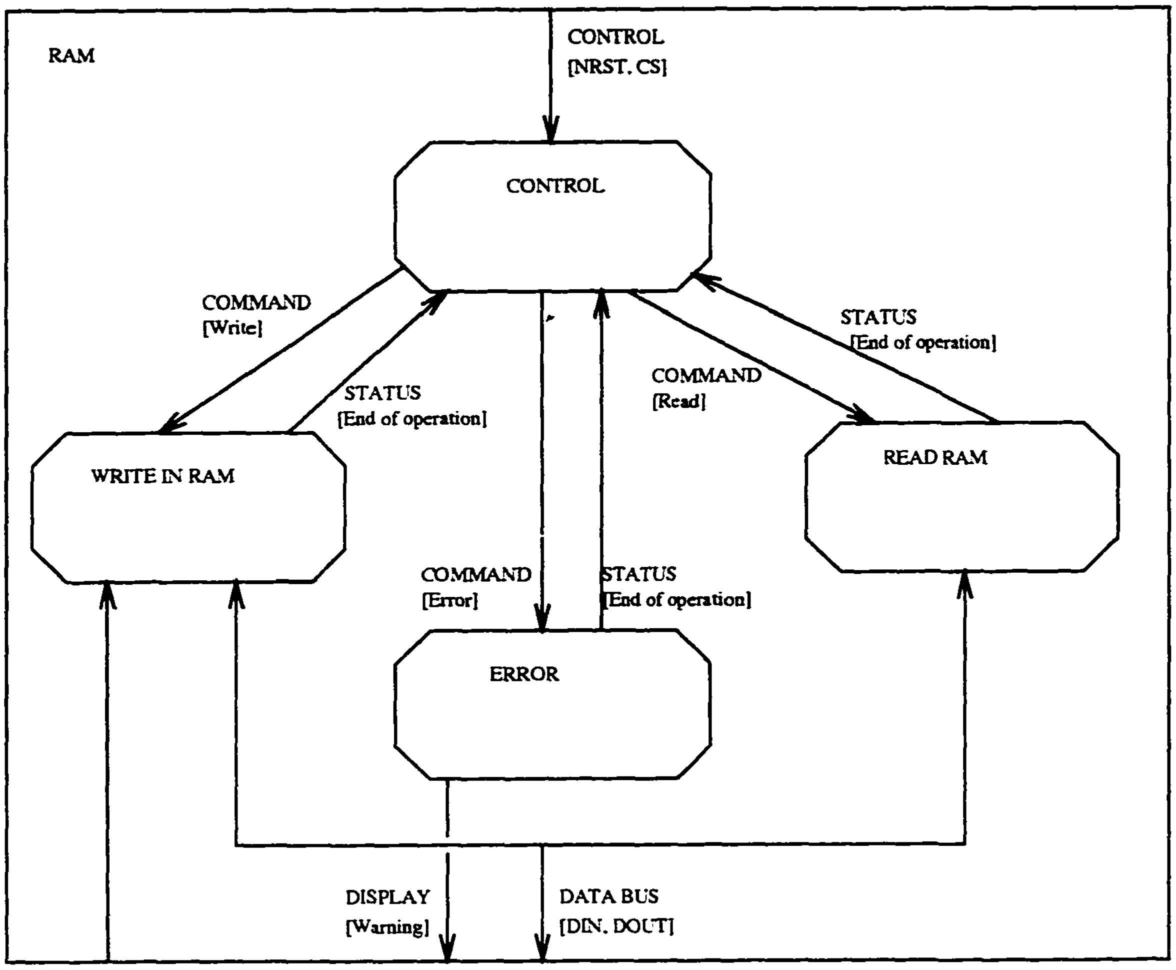


Figure 4-6: RAM Specification Using SDL

As an illustration for SDL, the specification of the RAM cell, described in Section 2.3, is shown in Fig. 4-6. For the purpose of showing a SDL description style, a partition is shown in Fig. 4-6 knowing that the RAM cell can be a leaf by itself because the RAM cell can be described using a state machine. So, as shown in Fig. 4-6, there are four processes: CONTROL, READ, WRITE and ERROR. The Control process manages the read and write operations performed by READ and WRITE processes and for simulation purposes, the Error process was added to display warnings when an unexpected combination occurs on WR and RD signals.

For this description model, the RAM cell is also partially described regarding the specifications given in Section 2.3. Indeed, it is not possible to specify the RAM delay specifications using the SDL formalism.

4.2.5 Evaluation of Process Control Based Methods

In Chapter 1, the automatic transition from concept level to system level is defined such that it embeds at least the characteristics from the set C. The properties of the above programming or control based methods can be mapped with the list of concept level characteristics as shown in table 4.1.

	C_{ASM}	$C_{PetriNets}$	C_{CSP}	C_{SDL}	C_{combi}	C_{min}	C
Sequentially Decomposable Activities	✓	✓	✓	✓	✓	✓	✓
Concurrently Decomposable Activities		✓	✓	✓	✓		✓
State Transitions	✓	✓	✓		✓	✓	✓
Immediate Mode Change				✓	✓		✓
Activity Completion	✓	✓	✓		✓	✓	✓
Delay Specification						✓	✓
Asynchronous Activities							✓
Design for { Test, manufacturing, etc }							✓
Multiple Model Representations							✓
Reusability							✓

Table 4.1: Characteristics of Programming or Control Based Methods

Table 4.1 highlights that each method is a specialized method. Notice that Petri nets and CSP contain the largest number of characteristics because they both address the problem of sequentially decomposable activities, concurrently decomposable activities, state transition and activity completion. Even though they have the same characteristics, a second level of comparison is possible. Indeed, the Petri nets model involves a dynamic graphical description of a system whereas CSP is a static textual description style.

For each checked characteristics, a mechanism is defined. We immediately notice that the Petri nets and CSP methods have an advantage over ASM because the set of characteristics in ASM is contained in the two former ones ($C_{ASM} \subset C_{PetriNets}$ and $C_{ASM} \subset C_{CSP}$). Indeed, ASM does not have the ability to concurrently decompose activities. This last remark implies that a description using ASM can be also described using Petri nets or CSP with a reasonable effort. On the contrary, Petri nets and CSP descriptions cannot be modeled with ASM without dramatic changes, which should lead to some modifications in the specifications.

In the case of SDL, its C_{SDL} shows that it is a complementary method in comparison with the other ones. In addition to the sequentially and concurrently decomposable activities, it also has the immediate change mode characteristic. This characteristic means that SDL has a built-in mechanism for emergency cases.

From Table 4.1, we can see that by using a multi-model representation description method, the combination of two of the four presented methods (ASM, Petri nets, CSP, SDL) can lead to a new description methodology having a characteristic set C_{comb1} greater than any of these methods i.e. if the new description method combines either $C_{comb1} = C_{Petrinets} \cup C_{SDL}$ or $C_{comb1} = C_{CSP} \cup C_{SDL}$, the cardinal of C_{comb1} is then 5 ($\#C_{comb1} = 5$) versus $\#C_{SDL} = 3$, $\#C_{Petrinets} = 4$ and $\#C_{CSP} = 4$.

Table 4.1 shows, for comparison purposes, the characteristics set C for the most advanced description methodology in the next generation of CAD tools. Notice that the four described methods are far from meeting the requirements of the next generation of CAD tools. So, a strategy to define this next generation of tools is to consider one characteristic which is not addressed by other description styles. The minimal configuration of the next generation of tools, as introduced in Section 1.1.2, is considered with an emphasis on a particular characteristic which is "Delay specification". Because this new generation of CAD tools have the goal of verifying the characteristics referred to as "Multi-model representation", the above minimum configuration does not need to verify the characteristics of C_{comb1} in addition to the Delay specification characteristic. Later, a combination of description styles will fill the gap in order to meet the requirement for a CAD tool having the characteristics C.

The next section reviews description methods from the microelectronics design methods introduced in Section 4.1 as the second family of description methods.

4.3 Microelectronics System Design Methods

This section focuses on description methods defined specifically for the microelectronics domain. This section reviews description methods in order to identify a good candidate to initiate the VHLLS with, at a minimum, the characteristics C_{min} defined in Section 1.1.2.

4.3.1 Hardware Description Languages

Hardware Description Languages (HDLs) such as VHDL [IEE93], Verilog^{TM} [TM91] and HardwareC [KM88] are used to describe hardware from the abstract to the architecture level and to be able to simulate, test, validate, and synthesize designs before implementation. They exhibit semantics common to high-level programming languages, such as data abstraction, behavioral operations, assignment statements and, control and execution ordering constructs to express conditional and repetitive behavior. The common denominator of these three HDLs is their software inheritance extended with hardware dependent features i.e. in VHDL a new category of variable types called *signal* is introduced. In general, specifications using these HDLs consist of a collection of concurrent *processes* which communicate with each other. Processes can be enclosed within a hierarchy of *blocks*. Blocks can be used to define structural relationships between the processes. A process specifies an algorithm as a set of sequential operations described in a manner close to a programming language such as C for HardwareC [KR78] and ADA for VHDL [LSU89].

A different philosophy has been taken in defining such languages. HardwareC is designed expressly to be a HDL for synthesis purposes whereas VHDL and Verilog^{TM} are simulation driven which gives them a more general syntax to describe a system. VHDL and Verilog^{TM} describe relationships between the inputs and the outputs of a system in terms of behavior, dataflow, structure or any combination thereof as illustrated in the example at the end of this section. The current progress on VHDL and Verilog^{TM} has restricted their semantic to synthesis's subsets which are commonly used in industry. Descriptions at the RT level in the behavioral domain (as defined in Chapter 3) can be synthesized with current tools as shown in Chapter 2. In chapter 6, VHDL is presented in more detail.

To illustrate HDL descriptions, VHDL code describing the RAM cell specified in Section 2.3 is presented. The code is composed of two parts. The first part is called *entity* and defines the interface of the RAM cell:

```
ENTITY RAM IS
          -- PORT describes the interface of the RAM cell
          -- An input is specified using the keyword IN
          -- An output is specified using the keyword OUT
          -- A type is associated with each signal: a bit or a word (bit_vector)
          PORT (
                NRST : IN bit;
                CS : IN bit;
                RD : IN bit;
                WR : IN bit;
                AD : IN BIT_VECTOR(0 TO 7);
                DIN : IN BIT_VECTOR(0 TO 3);
                DOUT : OUT BIT_VECTOR(0 TO 3);
                READY : OUT bit);
          -- To make the description easier to modify, constant values can be defined
          -- For this description, several durations are specified
          CONSTANT T_READY_U : time := 60ns;
          CONSTANT T_READY_D : time := 1ns;
          CONSTANT T_ACCESS : time := 40ns;
          CONSTANT T_WRITE : time := 5ns;
END RAM;
```

The second part, called *architecture*, is the description of the behavior of the RAM cell using algorithmic features:

```
ARCHITECTURE A OF RAM IS
-- Declaration of a constant which specifies the number of words the RAM can store
CONSTANT nb_words : INTEGER := 2 * *8;
-- Define the structure of the RAM: 256 4-bits words can be stored
TYPE type_memory IS ARRAY(0 TO nb_words - 1) OF BIT_VECTOR(0 TO 3);
-- Declare the variable which models the storage function of the RAM
SIGNAL M : type_memoire;
-- Definition of a function which convert the address into an index for M
-- In other words, it converts a bit string in an integer
FUNCTION value(bv : IN BIT_VECTOR) RETURN natural IS
VARIABLE n : natural := 0;
BEGIN
   FOR l IN bv'low TO bv'high LOOP
         n := n * 2;
         IF bv(l) =' 1'
             THEN
                     n := n + 1;
         END IF ;
   END LOOP;
   RETURN n;
END value;
-- Definition a type which enumerates the states the control function
-- of the RAM can be
TYPE Type_state IS (Init0, Init1, Waiting, RW, R, W, Err);
-- Declaration of a variable which represents the state the RAM controller is
-- Note that an initial value is predefined
SIGNAL state : Type_state := init0;
BEGIN
   -- Description of the RAM's controller
   -- The process is executed when one of signals NRST,
   --  CS, state, RD,WR changes
   main : PROCESS (NRST, CS, state, RD, WR)
   BEGIN --main
      -- The case statement allows to check in which state
      -- the controller is
      CASE state IS
              WHEN Init0 =>
                        -- The controller is in state INIT0
                        -- It checks NRST to become '0' in order to change
                        -- its state to INIT1
                        IF (NRST =' 0')
                            THEN
                                     state <= Init1;
                        END IF ;
              WHEN Init1 =>
                        -- The controller is in state INIT1
                        -- Next state is Waiting
                        IF (NRST =' 1')
                            THEN
                                     state <= Waiting;
                        END IF ;
              -- The controller is in state INIT1
-- Next state is RW
WHEN Waiting =>
        IF (CS =' 1')
            THEN
                    state <= RW;
        END IF ;
WHEN RW =>
        -- The controller is in state RW
        -- There is more than state. So, first the controller
        -- checks WR and RD to determine if the RAM is either in
        -- read mode implying the next state is R, or in write mode
        -- implying the next state is W, or else implying the next
        -- state is ERR
        IF (WR =' 0' AND RD =' 1')
            THEN
                    state <= R;
        ELSIF (WR =' 1' AND RD =' 0')
                THEN
                        state <= W;
                 ELSE
                        state <= Err;
        END IF ;
WHEN R =>
        -- The controller is in state R
        -- A ready pulse is sent
        -- the data read in memory is sent out
        Ready <=' 1' AFTER T_READY_U,'0' AFTER T_READY_U + T_READY_D;
        DOUT <= M(value(AD)) AFTER T_ACCESS;
        -- If a reset occurs within 1 ns time frame then the controller goes
        -- in state Init1 otherwise it goes to state waiting
        WAIT UNTIL (NRST =' 0') FOR 1ns;
                                            IF (NRST =' 0')
                                                THEN
                                                        state <= Init1;
                                                 ELSE
                                                        state <= Waiting;
                                            END IF ;
        WHEN W =>
                -- The controller is in state W
                -- a data is store in memory
                M(value(AD)) <= DIN AFTER T_WRITE;
                -- If a reset occurs within 1 ns time frame then the controller goes
                -- in state Init1 otherwise it goes to state waiting
                WAIT UNTIL (NRST =' 0') FOR 1ns;
                                                    IF (NRST =' 0')
                                                        THEN
                                                                state <= Init1;
                                                         ELSE
                                                                state <= Waiting;
                                                    END IF ;
                WHEN Err =>
                        -- The controller is in state ERR
                        -- A warning is issued and the next state is INIT0
                        ASSERT false
```

```
                                    REPORT "Wrong Value for WR and RD on rising edge of CS"
                                    SEVERITY warning;
                                    state <= Init0;
                        END CASE ; -- state
                  END PROCESS main;
            END A;
```

HDLs are important to investigate because they can meet a lot of requirement for VH-LLS. In fact, VHDL (the language of interest for the thesis) has the potential to perform some characteristics even though they are not a part of the language. For example, VHDL does not have the "immediate mode change" characteristic built-in it but with good programming skills, this characteristic can be implemented.

4.3.2 Silage

The Silage language [HIL85] was developed to address issues related to the specification of Digital Signal Processing (DSP) systems. DSP systems are easily conceived of as data-flow graphs, where a set of data values enters at the input nodes, computations are performed on them, and result values are delivered to the output node in the graph. Silage is essentially an *applicative* language in that it only specifies application of functions to manipulate a set of data values without having any variables or assignment operators.

The basic data objects in Silage are streams of value, called *signals*. Each Silage description has to have signals coming in and some signals going out. The same Silage description is then applied over and over again to the infinite sequence of input samples. In other words, an expression such as $(A + B)$ is composed of a stream of numbers denoted A and B as opposed to representing variables or array elements in conventional programming languages. A Silage program consists of a set of *definitions* which defines new values as a function of other values. As the assignment of signals represents a flow of data, the order of the definitions is not relevant. To refer to a signal in the previous sample interval, a delay operator is defined and noted :@. For example, $out = in + in@1$ semantically means $(\forall t : 0... + \infty) :: out(t) = in(t) + in(t-1)$ (at an instant t, the value of the output is equal to the sum of the value of the input at the same instant t and the input at the previous instant (t - 1)). Also multi-dimensional arrays of signals are possible. For that matter, operators such as *sum* or *max* are defined over an entire array. Other constructions can be used as well: conditional expressions to select one expression from a set of expressions based on guarded conditions, stream manipulation operators enabling up or down sampling of a signal, and macro expansion grouping a set of definitions. However, recursion or iteration constructions are not allowed in Silage. These constructions are not been defined in Silage.

Silage cannot be used to describe the RAM cell specified in Section 2.3 because the normal use of a RAM does not need a constant data stream to operate. On the contrary, the RAM reacts to control signals. Good applications for Silage are digital filters and other Digital Signal Processing (DSP) applications. For our purpose, it is interesting to talk about it because a system under specification could use features from DSP. So, a language like Silage should be a part of the set of description methods provided by the next generation of CAD tools.

4.3.3 SpecCharts

The SpecCharts language [VNG91a, VNG91b] consists of a hierarchy of states, represented in combined graphical and textual form, while catering to the expression of concurrent behavior and specification of constraints. This language combines the three aspects of system specification (control, behavior, and structure) into a single, unified environment. The concept of *behavior* is defined so as to describe a system with principles from Finite State Machines (FSM) and VHDL. A hierarchy notion, called *behavioral decomposition*, allows a decomposition of behaviors into either processes (also referred to as concurrent behaviors), or states (also referred to as sequential behaviors) which are sequenced by conditional arcs. At a leaf level in the hierarchy, a behavior uses VHDL sequential statements to specify actions the system needs to accomplish.

In SpecCharts, a box represents a behavior. A transition arc sequences sequential behaviors and a dotted line identifies concurrent behaviors. A feature devoted to managing a hierarchical language has been developed in order to respond to an external event: *hierarchical activation/deactivation* allowing a deactivation of any sub-behavior at any time. Another type of transition towards the next appropriate state is called *transition immediately* and gives the option exiting the current behavior, then suspending its execution. For cases other than the immediate transition, a mechanism is in place to flag a behavior which has completed its actions, allowing other states to be aware of that completion. This is called *behavioral completion.* Associated with that behavioral completion mechanism, a *transition on completion* arc causes a transition only when the source behavior has completed execution of its actions and the associated condition is true.

A translation process had been developed to generate a VHDL description from the SpecCharts language [VNG91c]. Templates are defined to map SpecCharts with a VHDL structure. Each behavior is translated into a block structure following the same hierarchy as SpecCharts i.e. a sub-behavior becomes a sub-block. Control statements defined in SpecCharts i.e. state activation/deactivation, are implemented as follows:

- a VHDL *wait* statement is sensitive to an activation by a parent of its behavior. During the activation mode, the resulting task is either activating/deactivating the proper sub-behavior or executing a VHDL code;
- during a deactivation mode, the behavior has to deactivate a sub-behavior either im-

mediately or after completion of actions depending of the type of transition requested;

- during an activation mode, the behavior is responsible for informing its parent upon completion of actions requested from it.

When an immediate transition occurs, a complete mechanism is activated to force current behavior and its sub-behavior to deactivate immediately and cease all signal assignments, thus preventing any signal assignments having the clause "*after* a time delay". The second type of transition is transition upon completion. In this case, it is verified that all statements within a behavior have been completed even if time delays are assigned to waveforms. So, a mechanism is implemented to evaluate the time of full completion of a behavior. Timing variables are introduced to measure the time spent during a wait statement (*global-time*) and to measure the remaining time necessary to complete all actions within the current behavior (*remain-time*).

The RAM specification is used to illustrate SpecCharts as shown in Fig. 4-7. This description is similar to a state machine where the initial state is marked with the dot extended with an arrow. The actions associated with each state are described in the box which model the state. In addition to the state machine, a declarative part is added at the top of the graph along with the name of the system. In Fig. 4-7, this declarative part contains the name RAM, the definition of the interface of the RAM cell and several constants.

Notice that SpecCharts has very advanced features such as a complex mechanism for its hierarchical structure because, for a characteristic like an "immediate mode change" the system under description must have all the processes stopped at once in the whole hierarchy as well as canceling signal assignment associated with duration (for example, we saw in Section 4.3.1 the signal "ready" was assigned with a waveform such that ready was

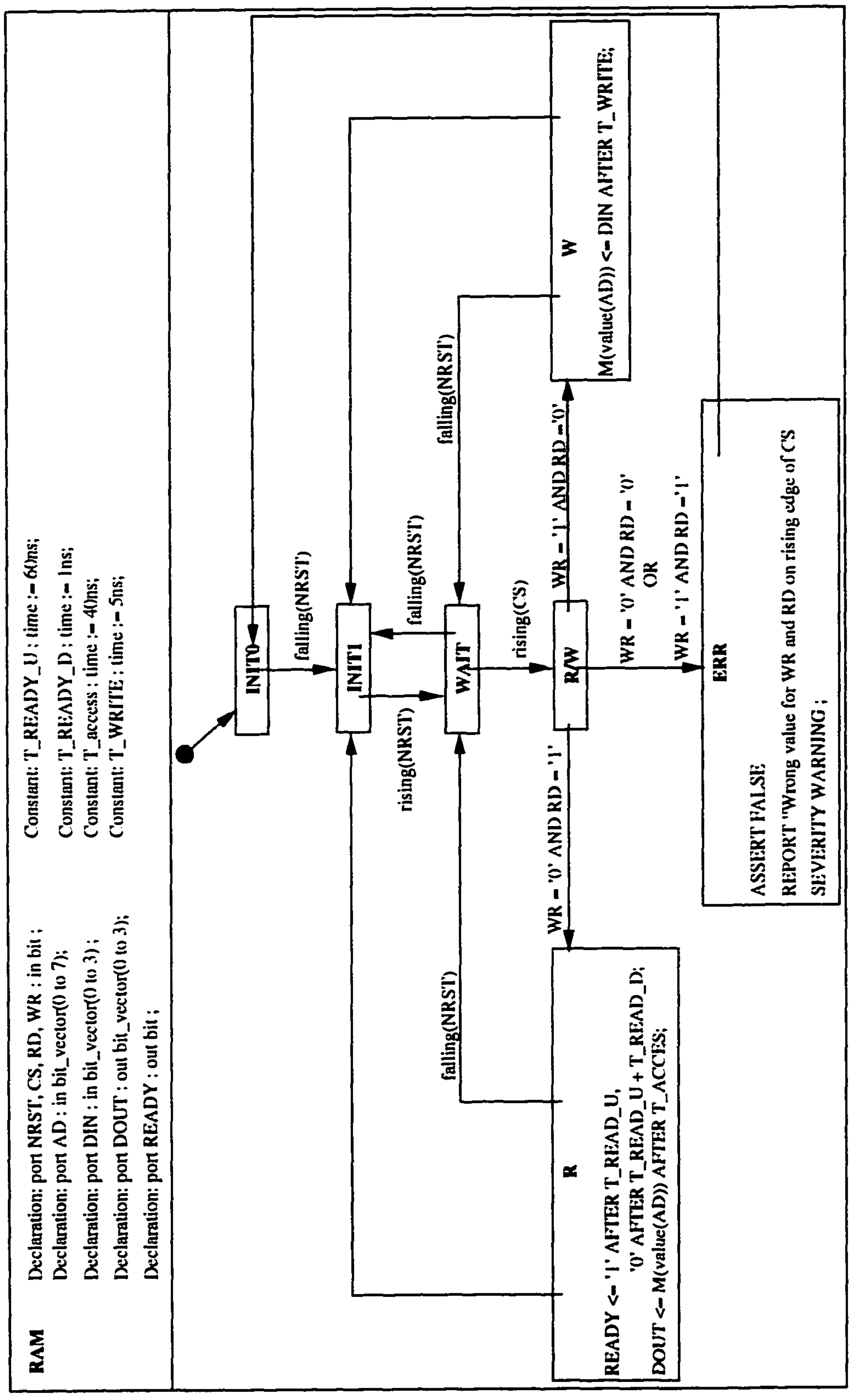

RAM
Declaration: port NRST, CS, RD, WR : in bit ;
Declaration: port AD : in bit_vector(0 to 7);
Declaration: port DIN : in bit_vector(0 to 3) ;
Declaration: port DOUT : out bit_vector(0 to 3);
Declaration: port READY : out bit ;
Constant: T_READY_U : time := 60ns;
Constant: T_READY_D : time := 1ns;
Constant: T_access : time := 40ns;
Constant: T_WRITE : time := 5ns;
INIT0
INIT1
WAIT
R/W
falling(NRST)
falling(NRST)
rising(NRST)
falling(NRST)
falling(NRST)
falling(NRST)
rising(CS)
WR = '1' AND RD = '0'
WR = '0' AND RD = '1'
WR = '0' AND RD = '0' OR WR = '1' AND RD = '1'
W
M(value(AD)) <= DIN AFTER T_WRITE;
R
READY <= '1' AFTER T_READ_U, '0' AFTER T_READ_U + T_READ_D;
DOUT <= M(value(AD)) AFTER T_ACCES;
ERR
ASSERT FALSE
REPORT "Wrong value for WR and RD on rising edge of CS"
SEVERITY WARNING ;

Figure 4-7: RAM Description Using SpecCharts

getting the value '1' after 60 ns and '0' again after 61 ns). In this case, if the system must leave the read state after 30 ns, the signal ready does not have the time to complete its assignment. So, SpecCharts has a mechanism to identify these waveform assignments and to cancel them.

4.3.4 State Action Tables

State-Action Table [HCG93] provides a concise tabular notation for state-based design descriptions, where the state sequencing of the design can be expressed clearly in a state table and the datapath operations can be expressed using textual assignment statements in each state.

In a state-action table, a column defines the type of the values on it or attribute of a state and a row establishes relationships between these typed values. Therefore, a state is characterized using a set of attributes:

- *PS* identifies the present state;
- *SCOND* is the condition for a transition to a next state;
- *NS* defines the next state;
- *ORDER* specifies the ordering of actions within a given state, stipulating a dependency between actions (known as *chaining*);
- *CV* is a list of conditions selecting proper actions to be executed;
- *ACOND* is the assignment condition for each action. The composition of these conditions involves asynchronous input signals, clock signals or boolean expressions;
- *ACTIONS* lists a sequence of operations required in the given state. This can be done

using functions as well as simple assignments. When using functions, operators can be used such as *operator pipelining* and *multi-cycle operators*. By nature a function may be composed of a sequence of operations which can take place over multiple time steps and may have multiple return values. So, these two operators allow operations to take one or more states to complete an operation (operator pipelining) and to partition a single operation into some number of sequential time steps:

- *AC#* allocates a unique identifier to each row:
- TIMING is an attribute which is decomposed into four sub-attributes specifying timing constraints:
  - *AB* (Action-Based constraint) defines a timing constraint on the action contained in the same row;
  - *SB* (State-Based constraints) defines the time needed for the considered state to have its actions finished;
  - *EB* (Expression-Based constraints) defines a timing constraint which has to be applied between two actions in the same state:
  - *TB* (Transition-Based constraints) defines a timing constraint which has to be applied for a transition between the present state and the next state.

A notion of hierarchy is also introduced in this methodology. Its main purpose is to represent multiple clock phases by classifying states and atomic actions.

As an illustration of the State-Action table description, we use again the case study described in Section 2.3. For each state of the RAM cell controller as shown in the column PS, a row is created. When one state has more than one next state, an new row is added accordingly as illustrated with state RW. A transition is controlled by a condition in column

PS	SCOND	NS	ORDER	CV	ASCOND	ACTIONS	AC#	AB	EB	SB	TB
INIT0	T	INIT1	0	(NRST = '0')	T		1				
INIT1	T	WAIT	0	(NRST = '1')	T		2				
WAIT	T	INIT1	0	(NRST = '0')	T		3				
		RW	0	(CS = '1')	T		4				
RW	T	R	0	(R='1' and W='0')	T		5				
		W	0	(R='0' and W='1')	T		6				
		ERR	0	((R='0' and W='0')) or (R='1' and W='1'))	T		7				
R	T	WAIT	0	(NRST = '0')	T	out ← Mem(addr)	8				8,1ns,2
W	T	WAIT	0	(NRST = '0')	T	Mem(addr) ← data	9				9,1ns,2
ERR	T	ERR	0	F	T		10				10,1ns,1

Table 4.2: RAM Description in State-Action Table

CV. Actions are specified in the column ACTIONS shown for the state R, W and ERR. For these three states, another information is contained in the table which is a time constraint applied on the transition from state R or W to INIT1 meaning that if a reset occurs, no matter what, the state change is effective after 1 ns. The time constraint as defined in this description model does not meet the characteristic referred to as "delay specification".

4.3.5 Evaluation of the Microelectronics Based Methods

In the previous section, we reviewed typical description methodologies from the microelectronics field. Based on the characteristic set defined in Section 1.1.2, a comparison is performed to evaluate which method would be the most appropriate to meet the requirement for the next generation of CAD tools. Table 4.3 shows the characteristic set for each of these methods as well as the desired characteristics C of the next generation of CAD tools and the minimal configuration C_{min} for the initial version of these future CAD tools.

Table 4.3 shows that SpecCharts meets a larger number of characteristics. SpecCharts was defined as an extension of VHDL. It follows a bottom-up approach, meaning that SpecCharts helps the designer to construct a VHDL code where a set of predefined structures

	C_{HDL}	C_{Silage}	$C_{SpecCharts}$	$C_{StateActionTables}$	C_{min}	C
Sequentially Decomposable Activities	✓	✓	✓	✓	✓	✓
Concurrently Decomposable Activities	✓	✓	✓			✓
State Transitions			✓	✓	✓	✓
Immediate Mode Change			✓	✓		✓
Activity Completion		✓	✓	✓	✓	✓
Delay Specification					✓	✓
Asynchronous Activities	✓					✓
Design for { Test, manufacturing, etc }						✓
Multiple Model Representations						✓
Reusability	✓					✓

Table 4.3: Characteristics of Microelectronics Based Methods

is provided. For example, describing a system behavior which contains a state machine is becoming a far easier task using SpecCharts than writing the VHDL. It helps minimize the chance of error during design. SpecCharts offer a lot more features than the commercial tools described in Chapter 2 because they embed more than just sequentially decomposable activities, state transitions and activity completion.

Like SpecCharts, the State Action Table description method is constructed following a bottom-up approach. These two methods have the same roots because they were developed in the same research laboratory (University of California, Irvine). State Action Tables are interesting because there is a mechanism to set time constraint which is to set the period of time the system has to stay in one state as opposed to the time encapsulation specified by the characteristic "delay specification" (it defines the maximum duration a system can stay in one state). However, these time constraints will be important to add in the formalism of the next generation of CAD tools.

The Silage description method is worth referring to because it is a common description method in microelectronics when the system has an important data stream to manipulate. It has been presented for informational purposes and additionally to stress that the next generation of CAD tools needs to provide this type of description method.

The last description method is HDL, with a particular emphasis on VHDL, because it is a more abstract description language than Verilog^{TM} and HardwareC. In addition, it has been an international standard since its creation in 1987. Table 4.3 presents HDLs as a description language that meets few of the characteristics. As a reminder, a characteristic is checkmarked when a mechanism is built in the description method to perform this characteristic. In the case of VHDL, even though very few characteristics are built-in, the syntax is flexible enough to allow a VHDL description of them. This is indeed a reason, SpecCharts and State Action Tables are defined upon VHDL and are able to out-shine most of the description methods in this chapter. However, the main disadvantage of VHDL is its absence of visual representation. Most hardware designers like to "see" a design database rather than have it as a textual string. Also, a VHDL description can become cumbersome very quickly.

As seen in the previous section and this one, no method meets the requirement for the minimal configuration of the next generation of CAD tools (moreover the next generation of CAD tools which meet the characteristics of C). Also, not all the presented description methods have the ability to automatically generate a description in the behavioral domain (any levels) such as CSP, Silage. On the contrary, SpecCharts and State Action Table descriptions provide a description method which is captured at the system level in the behavioral domain, and an evolution process (as defined in Chapter 3) allows an automatic translation of their descriptions into the architecture level in the behavioral domain using VHDL. Thereafter, a composition of evolutions automatically generates the design in a targeted technology.

The next chapter introduces one approach for meeting the minimum requirement for the next generation of CAD tools and implementations of the evolution process defined in Chapter 3 as VHLLS. Indeed, with the addition of more characteristics, the abstraction of the descriptions increases.

Chapter 5

VHLLS Design Strategies

In Chapter 1, the notion of VHLLS has been introduced and characterized. In addition, motivations for developing a such method have been outlined. This leads to a minimal configuration, C_{min}, for the next generation of CAD tool referred to as SPECIAL. Chapter 2 contrasts a conceptual view of VHLLS with commercial tools in order to give a preliminary definition of VHLLS. Chapter 3 formalizes the design space as well as VHLLS. Chapter 4 reviews the most significant description methods which have been considered as a VHLLS methodology. As a result, no methods satisfied the requirement of VHLLS as stated in Chapter 1.

The goal of this chapter is to propose a framework for the implementation of the VHLLS methodology. To achieve this, two different strategies are investigated. As mentioned in the previous chapters, the VHLLS methodology is an evolution from the concept level in the behavioral domain to the system level in the behavioral domain. At the concept level, graphical methods are preferred because they are more widely used in the engineering field and they can carry more information than text. This does not mean that natural language should not be part of the input description, and SPECIAL has a potential of addressing

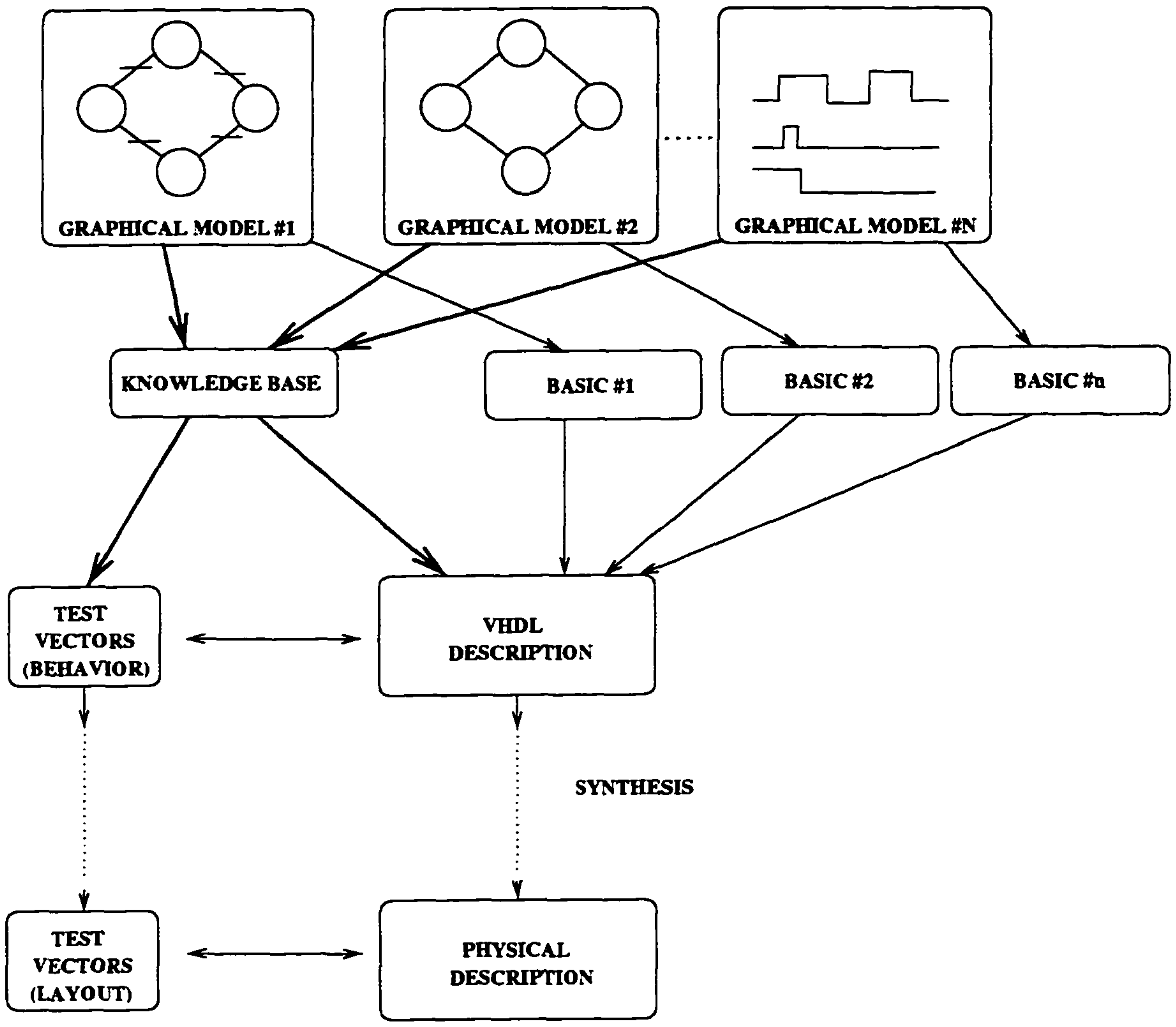


Figure 5-1: Global Strategy: Specification-Behavior Synthesis

this requirement but this type of description is not considered in this thesis. The system level description in the behavioral domain uses the most popular HDL called VHDL, as introduced in Section 4.3.1. VHDL will be described in more details in Section 6.1.

The VHLLS methodology is performed in two steps: (1) concept synthesis, and (2) concept refinement (both introduced in Section 3.3). To reduce the translation complexity, an intermediate representation is introduced. Thus, the VHLLS processes can be viewed as shown in Fig. 5-1.

In this figure, the two strategies are represented and are as follows:

- *Basic VHLLS:* for a given graphical representation, a specific translator can be created to generate the corresponding behavioral description;
- *Advanced VHLLS:* the necessary knowledge is extracted from each graphical representation and put into a unified model. From this unified model, a unique translation process is performed to generate the corresponding behavioral description.

In Fig 5-1, the upper boxes, referred to as graphical models, represent description models at the concept level in the behavioral domain. The other boxes can be viewed as transfer functions. The basic VHLLS strategy is represented by the link between a graphical model and a VHDL description through an intermediate representation shown as a box labelled "BASIC" followed by a number. Notice that for each graphical representation, one intermediate representation is necessary justifying a unique label for each box. The advanced VHLLS strategy is represented by the link between a graphical model and a VHDL description through an intermediate representation shown as a box labelled "KNOWLEDGE BASE". This link has the property that all the graphical models lead to the same box and only one output of this transfer function is needed to generate the corresponding VHDL description.

To define these two strategies, a sub-set of the possible descriptions is chosen to meet the characteristics of the first generation of VHLLS processes as specified in Chapter 1. Indeed, the minimum set of characteristics, referred to as C_{min} in Chapter 1 and shown in Table 5.1, is applied to define a graphical model which is presented in Chapter 6. This description is referred to as a *pseudo-state diagram* because it uses the principle of a conventional state diagram where time is encapsulated. To implement this minimal configuration of VHLLS starting from the pseudo-state diagram, we must define the meaning of the characteristic **Delay Specification** also referred to as *time encapsulation*. As a consequence of this

C_{min} = {

Characteristics	Checkmark
Sequentially Decomposable Activities	✓
Concurrently Decomposable Activities	
State Transitions	✓
Immediate Mode Change	
Activity Completion	✓
Delay Specification	✓
Asynchronous Activities	
Design for { Testability, Manufacturing, etc }	
Multiple Model Representations	
Reusability	

}

Table 5.1: First Generation VHLLS Characteristics: C_{min}

characteristic, the formulation of conditions in the context of the pseudo-state diagram must be examined. Therefore, Section 5.1 defines the notion of time and some fundamental notions about facts and events, and features associated with time encapsulation. Section 5.2 presents the basic VHLLS strategy whereas Section 5.3 presents the advanced VHLLS strategy.

5.1 Fundamental Definitions

Before presenting the two strategies for implementing VHLLS, we need to introduce some fundamental definitions about the representation of time. Thereafter, we define the meaning of events and facts which are fundamental for the representation of time for the VHLLS process.

5.1.1 Model of Time

As mentioned earlier, we need to address the representation of time in the VHLLS process in order to encapsulate time. This process is accomplished by introducing a formal representation of time. Furthermore, this approach involves constructing representations and logical systems to handle time. The framework for handling time is in our case a temporal logic[1].

In order to specify a temporal logic in a semantical manner, the following list of items is defined [GHR93]:

1. the time flow of the logic;
2. the units of time needed to determine truth values;
3. the temporal connectives used;
4. the truth conditions for the connectives.

In order to elaborate on each of these elements from the above list, first a temporal structure needs to be introduced. Let T be a discrete time set. For any t_1 and $t_2 \in T$, there exists an ordered relationship between t_1 and t_2 denoted $<$ such as $t_1 < t_2$ means t_1 is before t_2. Let $t_0 \in T$ be the first point of discrete time. In the temporal logic terms, a logical expression referred to as a proposition p can become true at an instant $t_0 \in T$. An assignment function h is also included in a temporal structure to define either operators or predicates. Formally, the temporal structure can be defined as follows:

Definition 5.1 *A* ***temporal structure*** *has the form $(T, <, t_0, h)$, where T is an indexed set, $(T, <)$ is a flow of time, $t_0 \in T$, and h is an assignment.*

In the above structure, the notation $(T, <)$ is a generalization of time space without constraints imposed by t_0 and h, and $<$ is a binary relation within the indexed set T. Temporal structure, $(T, <, t_0, h)$ becomes time domain for proposition p if and only if $\|p\|_{t_0}^{h} = 1$

[1]Temporal logic theory is an extension of the logic theory with time sensible operators and predicates.

which means the proposition is true at time t_0 and through out the assignment h. p is said to be valid in $(T, <, h)$ if $\forall t \in T, \|p\|_t^h = 1$. p is said to be valid in $(T, <)$ if and only if any h, p is valid in $(T, <, h)$.

From the above discussion and the above temporal logic list, we have:

1. the time flow which can be interpreted as a continuous increase of an index $t(i) \in T$ such as $t(i-1) < t(i)$;

2. the unit of time follows the international unit of time i.e. seconds,milliseconds, microseconds, etc...

3. the temporal connectives represent temporal operators or predicates. By default, the eligible standard logical operators are $\wedge$ (and), $\vee$ (or), $\neg$ (not), etc...

4. The truth conditions determine logical value of temporal expressions. The following notation is used for predicates with temporal expressions. Predicate S is defined as $S(A, B)$, where A, B are propositions, and interpreted depending on the combination of temporal characteristics of A and B. Formally, a connection between temporal characteristics of A and B, and predicate $S(A, B)$ is accomplished by using assignment h representing the equality "=". For example, if both A and B are true in the sense that in time B is always true whenever A is true then $S(A, B)$ = "A and B are true in the sense that in time B is always true whenever A is true". In a more formal manner, the truth conditions for this connective can be defined as follows:

   $S(A, B)$ is true at n ("now") if for $t < n$, A is true at t and for all points between t and n, B is true.

Having introduced the general definition of the time flow, we can now introduce the model of time for a VHLLS methodology. In the microelectronics domain, the time flow

Time Domain	Definitions	Comments
$(T, <)$	Generalization of the structure of time	also known as time flow
$(T, <, h)$	Structure of time under the assignment h	the time flow is associated with an assignment mechanism which allows the definition of temporal expressions
$(T, <, t_0, h)$	Structure of time	It is a constraint version of $(T, <, h)$ meaning that the assignment definitions of temporal expressions are effective only at the instant t_0
$(T, <, =)$	Structure of time under the assignment "="	Same as $(T, <, h)$ with h = "="
$(T, <, =, T_a)$	Structure of time which embeds branching future	Same as $(T, <, h)$ with h = "=" and in addition, the notion of future is embedded as well as its uncertainty with a non-unique value of future instant a

Table 5.2: Time Domains

can be characterized as follows: there is a single time path going from a system state in the past to the one in the present. For example, a RAM cell is in an idle state waiting for a chip select signal. As soon as the chip select signal is active, the RAM cell switches to fetch mode. Considering the instant when the chip select signal becomes active as a reference point, referred to as *present* or *now* (noted n), the *past* of n ($\forall t \in T$ *such as* $t < n$) is linear because all the states of the RAM are known. However, future system states are not predictable in terms of both time and state, and depend strongly on events coming from the external world to the system. Such an interpretation of the future can be modeled with the notion of *branching future*. In other words, taking the same example as above, the state of the RAM cell after n (now) ($\forall t \in T$ such that $t > n$ represents the *future*) cannot be identified. So, depending on possible events, the RAM state will be in one state or another. The potential of future behaviors are modeled using the notion of branching future also referred to as planned future. This flow of time can be modeled using a linear representation[2] of time $(T, <, =)$ adding the future branching representation T_a such as $(T, <, =, T_a)$, $a \in T$. In Fig. 5-2, the time increases linearly when going from the left to the right. The planned future behaviors are represented by broken line i.e. branches, for instance, they occur at time instants t and s. So, in the first case, $a = t$ and temporal structure $(T, <, =, T_t)$ is obtained. For the second case, $a = s$ the temporal structure $(T, <, =, T_s)$ is defined. With this notation, T is the set of moments of time, and $<$ is an irreflexive and transitive relation within T and $a \in T$. An illustration is given in Fig.5-2. Axiomatically, $(T, <, =, T_a)$ has the following properties:

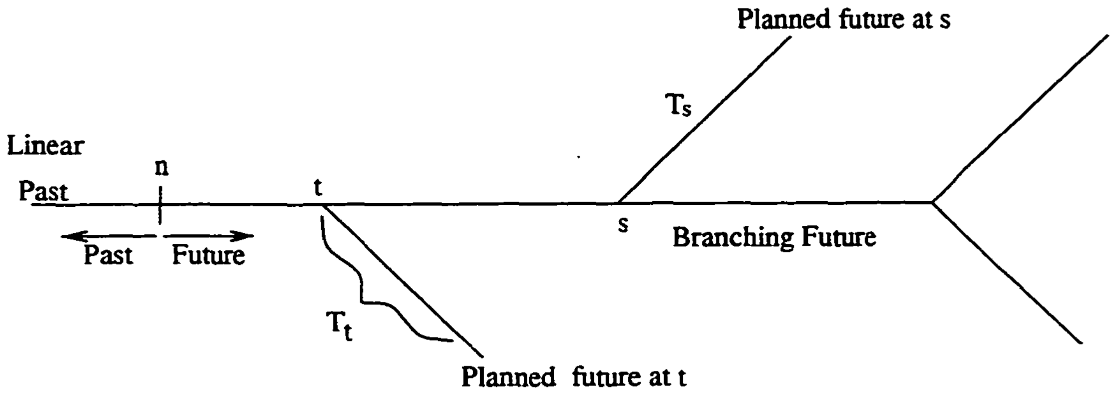


Figure 5-2: Branching Future

1. $<$ is irreflexive and transitive, i.e.

   (a) $\forall x \in T, \neg(x < x)$

   (b) $\forall x, y, z \in T, (x < y \wedge y < z \Rightarrow x < z)$

[2]linear representation is interpreted as a straight line which models the constant evolution of time (same pace).

2. The past is linear:

   $\forall x, y, z \in T, (x < z \wedge y < z \Rightarrow x < y \vee y < x \vee x = y)$

3. for each branch in Fig. 5-2 generalized by the corresponding T_a, the following properties hold:

   (a) $a \in T$

   (b) $T_a \subseteq T$

   (c) $a \in T_a$

   (d) $(T_a, <)$ is a linearly ordered flow of time

   (e) $\forall x, y, z \in T, (x \in T_a \wedge x < y \wedge y < z \wedge z \in T_a \Rightarrow y \in T_a)$

   (f) $\neg \exists x \in T \textit{ such as } ((\forall y \in T_a, (x < y)) \wedge (\forall y \in T_a, (y < x)))$

4. Note that **for any** $x \in T$ the past of x is the actual history but the future may be branching and unknown.

Therefore, a notion of time is defined allowing a representation of the time flow, an illustration of the past, and the present, and an uncertainty of the future. This uncertainty is dependent on unpredictable changes modeled by the branching future. It has to be stressed that the conclusions were derived from a formal model of time, and not using casual perception of the reality.

In the VHLLS process, a corresponds to the instant where the system changes its state. T_a corresponds to time interval $T_a = [a_1, a_2]$ where the system is in a certain state where a_1 is known and represents the instant when the system changes its state. a_2 remains unknown.

5.1.2 Facts and Events

As stated in the previous section, temporal connectives are specified in the defined model time. For the purpose of this research, two major notions: *fact* and *events* are formally clarified and analyzed. Intuitively, a general interpretation [SOW84] of an event might be as follows: events are the means by which agents (i.e. input signals) classify certain useful and relevant patterns of change. Another interpretation from Goldman [GOL70] is that in a common sense, an event corresponds to a change in an element, caused or partially caused by a stress. Definitions proposed by Allen and Ferguson [AF94] assume that knowledge representation of events and facts can be effectively partitioned into two types of formulae:

- *event formulae* state that something happened that (possibly) resulted in a change;
- *fact formulae* represents everything else, but typically describe some properties of the universe (possibly temporally qualified).

A representation of events can be performed using time intervals included in T. In particular, events occur over intervals of time, and cannot be reduced to some set of valid properties (holding true) at one instant [AF94]. Therefore, an event occurs in an indivisible time interval (Let I, I' be two time intervals. An event occurring over interval I implies that there exists no interval I' such as $I' \subset I$). That indivisible time interval is referred to as an *instantaneous interval* as opposed to a *time interval* (or duration). The set of instantaneous intervals is noted Ξ and the set of time intervals is noted Υ.

The temporal logic introduced in Section 5.1.1 is classified as a first-order predicate calculus[3] which contains several categories. The following four items are the basic categories for modeling in this type of logic:

- TIME-REPRESENTATION (GTS standing for Global Time Set) being $GTS = \Xi \cup \Upsilon$

[3]First order predicate calculus uses first order variables such as x, y, etc... Second order predicate calculus uses second order variables such as $\Phi(x)$ where $\Phi(x)$ is any formula in this logic.

where Ξ is the set of instantaneous intervals and Υ the set of time intervals;

- PROPERTY for denoting propositions;
- OCCURRENCE for modeling modifiers and qualifiers of events as predicates acting on temporal expressions;
- TEMPORAL EXPRESSION themselves.

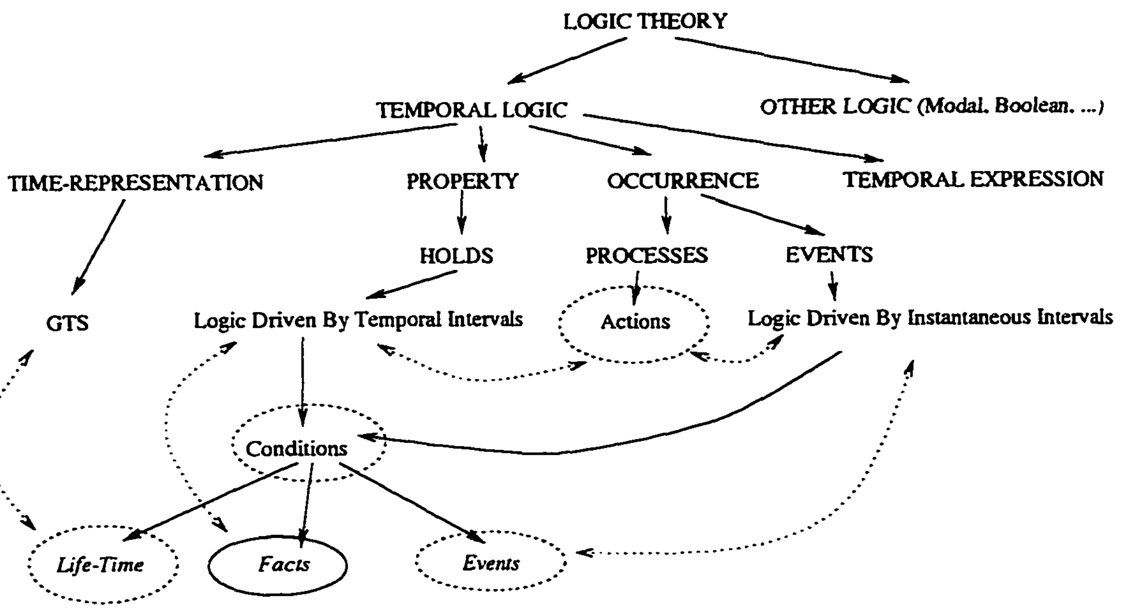


Figure 5-3: VHLLS Temporal Logic Hierarchy

An important predicate for PROPERTY is the predicate HOLDS which asserts that a property p holds (i.e. is true) during time interval I. Thus, HOLDS(p, I) is true if and only if property p is true during I. Another important type is the type OCCURRENCE. Indeed, the OCCURRENCE type is divided into two subtypes, *processes* and *events* as illustrated in Fig. 5-3. Note that the purpose of Fig. 5-3 is for clarification of notion introduced here. It should be noted that TIME-REPRESENTATION and TEMPORAL EXPRESSIONS are affecting all categories in the depicted Fig. 5-3. Finally, subtrees with dotted circles in Fig.

5-3 depict class hierarchy. So, another element of Fig. 5-3 called Processes refer to activities not involved in a culmination or anticipated results. Events describe activities that involve a product or outcome. Using the above notions, a characterization of these two subtypes of OCCURRENCE over the set of time representation GTS is:

- the set of intervals from the *event* subtype (see Fig. 5-3 contains indivisible intervals. In other words, an event occurs over the smallest time interval possible (i.e. $i \in \Xi$). This interpretation is consistent with the definition of an event introduced before.
- the combined features of *events* and PROPERTY(IES) where the PROPERTY type is defined as follows: if a proposition is true over an interval I then for all sub-interval I' ($I' \subset I$), a property holds over I'.

There are two main notions introduced: Conditions and Actions (as indicated by dotted circles in Fig. 5-3. In the *condition* class, three sub-classes are identifiable: *Life-Time*, *Facts* and *Events* (*Life-Time* in Fig. 5-3). Before going further in the description of that condition class, a basic set of mutually exclusive primitive relations that can hold between temporal intervals is introduced. Each of these relations is represented by a predicate in the TEMPORAL LOGIC. These relationships[4] with time intervals $I_1, I_2 \in \Upsilon$ are:

- $DURING(I_1, I_2)$ = "time interval I_1 is fully contained within I_2";
- $STARTS(I_1, I_2)$ = "time interval I_1 shares the same beginning as I_2, but ends before I_2 ends";
- $FINISHES(I_1, I_2)$ = "time interval I_1 shares the same end as I_2, but begins after I_2 begins";

[4]Note that the relationships follow the notation of truth conditions $S(A, B)$ = "statement".

- $BEFORE(I_1, I_2)$ = "time interval I_1 is before interval I_2, and they do not overlap";
- $OVERLAP(I_1, I_2)$ = "time interval I_1 starts before I_2, and they overlap";
- $MEETS(I_1, I_2)$ = "time interval I_1 is before I_2, but there is no interval between them, i.e., I_1 ends where I_2 starts";
- $EQUAL(I_1, I_2)$ = "time interval I_1 and I_2 are the same".

Including the inverse of each of these relationships (in the same order as in the list: $INCLUDES$, $STARTED-BY$, $FINISHED-BY$, $AFTER$, $OVERLAPPED-BY$, $MET-BY$), there are a total of 13 relationships between intervals as shown in Table 5.3. These are referred to as the *Allen's classification*[AF94].

By relating the sub-classes of the Condition class from Fig 5-3 to the Allen's classification, a Fact is a PROPERTY type, an Event is an EVENT type and a Life-Time is a TIME-REPRESENTATION type. A life-time is the maximum duration that a system can stay in a certain state. An action corresponds to a PROCESS type. Using the Allen's classification, a definition of these notions can be formulated.

Definition 5.2 *A fact is interpreted as a temporal predicate F(p, I). This predicate becomes true if and only if the proposition p is true over the whole interval I. Therefore, let p be a proposition and $I \in \Upsilon$ be a temporal interval such as:*

$$\begin{aligned}
(\mathrm{F(p, I)} \quad \Leftrightarrow \quad & \\
& \forall I', I'' \in \Upsilon, \\
& (1)\ BEFORE(I', I) \wedge BEFORE(I, I'') \\
& (2)\ \wedge HOLD(\neg p, I') \\
& (3)\ \wedge HOLD(\neg p, I''))
\end{aligned}$$

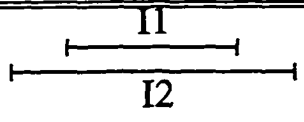
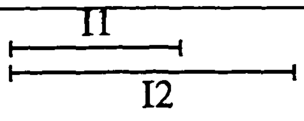
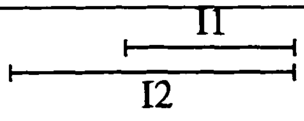
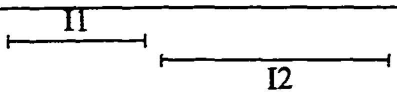
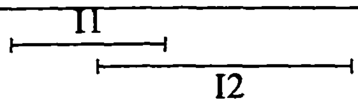
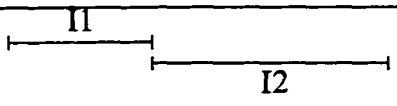
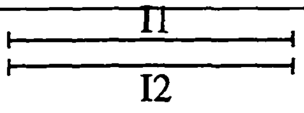
Temporal Relationships	Interpretation	Inverse Relationships
DURING(I_1, I_2)		INCLUDES(I_2, I_1)
STARTS(I_1, I_2)		STARTED-BY(I_2, I_1)
FINISHES(I_1, I_2)		FINISHES-BY(I_2, I_1)
BEFORE(I_1, I_2)		AFTER(I_2, I_1)
OVERLAP(I_1, I_2)		OVERLAPPED-BY(I_2, I_1)
MEETS(I_1, I_2)		MET-BY(I_2, I_1)
EQUAL(I_1, I_2)		EQUAL(I_2, I_1)

Table 5.3: Temporal Relationships

Definition 5.3 *An* ***event*** *is interpreted as a temporal predicate $e(p, i)$. This predicate becomes true if and only if the proposition p happens over instantaneous interval i. Therefore, let p be a proposition and $i \in \Xi$ such as:*

$$
\begin{aligned}
&(e(p, i) \Leftrightarrow \\
&\quad \forall i', i'' \in \Xi, \\
&\quad (1)\ BEFORE(i', i) \wedge BEFORE(i, i'') \\
&\quad (2)\ \wedge OCCUR(\neg p, i') \\
&\quad (3)\ \wedge OCCUR(\neg p, i''))
\end{aligned}
$$

Definition 5.4 *A* ***life-time*** *is a time interval $I_d \in \Upsilon$ associated with each state of a system. Interval I_d determines the duration the system can check conditions in order to change its* *state. When the time reference n (now)*[5] *passes the upper limit of this time interval, the system has to change to a predefined state.*

For the purpose of modeling a pseudo-state diagram as defined in the previous section, an event type predicate as shown in Fig. 5-3 is introduced. The main rationale for this predicate is to express the transition condition from a state s_i to s_j noted $cond_{s_i}^{s_j}$.

Definition 5.5 *A* ***transition condition predicate*** *denoted $cond_{s_i}^{s_j}(p, i)$ is true when the proposition p holds at $i \in \Xi$ or using the above notation:*

$$cond_{s_i}^{s_j}(p, i) = \text{"proposition } p \text{ holds at } i \in \Xi\text{"}$$

Another predicate needed is derived from the PROPERTY type as shown in Fig. 5-3 is defined to characterize the snapshot of the system during time interval I and denoted $STATE(I, s_i)$.

Definition 5.6 *A* ***state predicate*** *denoted $STATE(I, s_i)$ holds when the system is in state s_i during $I \in \Upsilon$ or using the above notation:*

$$STATE(I, s_i) = \text{"the system is in state } s_i \text{ during } I \in \Upsilon\text{"}$$

As a result, a change from one state to another can be formalized with the model of this transition expressed using the above temporal formalism. Let $m, n \in \aleph$, $I, I' \in \Upsilon$ be natural numbers and time intervals respectively. Furthermore, the following expressions for transition conditions can be stated:

1. the transition condition is a fact if and only if

$$\exists i \in \Xi, STATE(I, s_n) \wedge F(I', p) \wedge IN(I, i) \wedge IN(I', i) \Rightarrow \exists m, cond_{s_n}^{s_m}(p, i).$$

[5] *Refer to Fig. 5-2.*

The predicate $IN(I, I')$ is defined as:

$$IN(I, I') = STARTS(I, I') \vee DURING(I, I') \vee FINISHES(I, I')$$

2. the transition condition is an event if and only if

$$\exists i \in \Xi, STATE(I, s_n) \wedge e(p, i) \wedge IN(I, i) \Rightarrow \exists m, cond_{s_n}^{s_m}(p, i)$$

3. the condition is a life-time of the state s_n if and only if

$$\exists i \in \Xi,$$

$$STATE(I_d, s_n) \wedge MEETS(I_d, I') \wedge OVERLAPS(I_d, i) \wedge OVERLAPS(i, I')$$

$$\Rightarrow \exists m, cond_{s_n}^{s_m}(p, i)$$

4. the transition condition is a composition of facts and events when the following is valid. In this case, a new generic predicate is introduced to represent a fact or an event. Using the Global Time Set (GTS), introduced earlier, this new generic predicate is: $Excit_{s_i}(I, I', p, t)$.

Definition 5.7 *let $I, I', t \in GTS$ and p be a proposition, $Excit_{s_i}(I, I', p, t)$ is defined as:*

$$Excit_{s_i}(I, I', p, t) = STATE(I, s_i) \wedge \left\{ \begin{array}{c} F(p, I') e(p, I') \end{array} \right\} \wedge IN(I, t) \wedge IN(I', t).$$

We note that for an event described just above in this list (item 2), the condition

$IN(I', t)$ is redundant because in this case, by definition, I' is equal to t and then $IN(I', t)$ is always true. Using this new predicate, a list of transition conditions can be defined:

(a) COMPMULT:

$$\exists t_1, t_2, t \in GTS,$$

$$Excit_{s_i}(I, I_1, p_1, t_1) \wedge Excit_{s_i}(I, I_2, p_2, t_2) \wedge EQUALS(t_1, t) \wedge EQUALS(t_2, t)$$

$$\Rightarrow \exists j, cond_{s_i}^{s_j}(t, p_1 \wedge p_2);$$

(b) COMPADD:

$$\exists t_1, t_2 \in GTS,$$

$$Excit_{s_i}(I, I_1, p_1, t_1) \vee Excit_{s_i}(I, I_2, p_2, t_2)$$

$$\Rightarrow \exists j, (cond_{s_i}^{s_j}(t_1, p_1 \vee p_2) \wedge t_1 < t_2) \vee (cond_{s_i}^{s_j}(t_2, p_1 \vee p_2) \wedge t_1 > t_2);$$

(c) PIPE (sequential operations):

$$\exists t_1, t_2, t \in GTS,$$

$$Excit_{s_i}(I, I_1, p_1, t_1) \wedge Excit_{s_i}(I, I_2, p_2, t_2) \wedge (AFTER(I_2, I_1) \vee MEETS(I_1, I_2))$$

$$\Rightarrow \exists j, cond_{s_i}^{s_j}(t_2, p_1 \mid p_2).$$

In this section, fundamentals have been introduced and defined allowing a better understanding of the notion of time. Time modeling is crucial to meet the characteristic referred to as "Delay specification" introduced in Chapter 1. As a consequence, operators or predicates must be defined to encapsulate time. Having defined these notions, the next two sections focuse on the two strategies presented in the introductory section of this chapter extensively using the above notions.

5.2 Fundamental VHLLS Design Methodology

This section presents the first strategy to implement the VHLLS process. It is based on the mathematical model defined by Zeigler [ZEI84]. The principle of this strategy is to perform a translation process from the concept level in the behavioral domain into the system level in the behavioral domain as specified in Chapter 3.

5.2.1 Extended Zeigler Formalism

When we make specifications with a pseudo-state graph, we must consider having variables which we refer to descriptive variables of the system. They compose a set of variables characterizing the system. In this variable set, two types of variables exist: *input variables* and *non-input variables.* An input variable can be modified only out of the system. A non-input variable can be of two types : *state variables* and *non-state variables.* A subset of state variables characterize one state. A non-state variable is a set of descriptive variables not included in the other sets. The state variables allow the identification of the future system state. The non-state variables are for computing purposes at a given instant. These statements can be represented as follows:

$$\mathit{descriptive\ variables}(V_d)\left\{\begin{array}{l} \mathit{input\ variables}(V_i) \\ \left.\begin{array}{l} \mathit{state\ variables}(V_s) \\ \mathit{non}-\mathit{state\ variables}(V_{n-s}) \end{array}\right\} \mathit{non}-\mathit{input\ variables}(V_{n-i}) \end{array}\right.$$

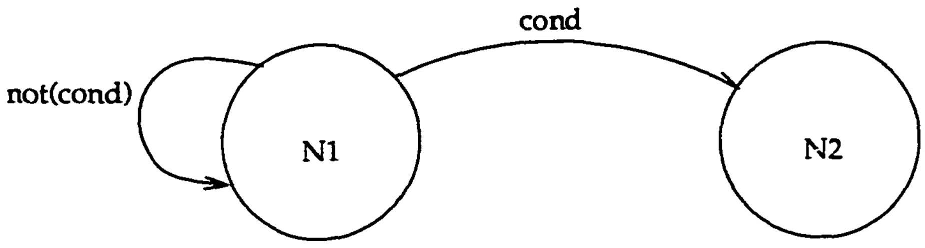


Figure 5-4: Waiting Interpretation

or in a more mathemical form as: $V_d = V_i \cup V_{n-i}$ and $V_{n-i} = V_s \cup V_{n-s}$. The transition from one state to another is modeled by a transition function which is a function of inputs and state variables such as:

$$\tau_z : S \times V_i \rightarrow S$$

The principle of τ_z is for the evaluation of the present state variables associated with the input variables to compute the future state. At this stage of the representation, we must express the action of waiting in a state when, at a given instant, no transition conditions hold. We represent this action with a transition from a state to itself with a transition condition which is a complement of all transition conditions applicable for the current state. This allows us to continuously poll the transition condition until one holds thus implying a system change into the future state. This type of loop is called a waiting loop.

The introduction of an output function allows a mapping from each state to a set of actions:

$$\psi_z : S \times V_i \rightarrow V_{n-s}$$

.

After introducing the fundamentals to model the pseudo-state diagram, we must consider the notion of time where the system is forced to leave a state. Being inspired by the Zeigler theory [ZEI84], we modify the transition function into two sub-functions. The first one is the external transition function τ_{ext} which has the same behavior as the function defined above τ_z. The second one is named internal function τ_{int} which uses the notion of state life-time I_d as defined in section 5.2. When the time index (t) is in I_d , the state change depends on state conditions. As soon as t is not in I_d the internal transition is applied. I_d can be expressed as $[t_{in}(s), t_{in}(s) + t_a(s)]$ where t_{in} identifies the instant the system enters $s \in S$ and t_a is a function which associates a life-time to the same state s. So when $t > t_{in}(s) + t_a(s)$, an automatic state transition is applied. Therefore, the function t_a associates a duration with each state of the system allowing the definition of its life-time: $t_a : S \rightarrow TIME$ where $TIME$ is defined as being a positive natural number associated with a time unit (for example, $1ns \in TIME$). These two transition functions are:

1. $\tau_{ext} : S \times I \rightarrow S$ where τ_{ext} is applicable when the current time index t verifies DURING(t, I_d)
2. $\tau_{int} : S \rightarrow S$ where τ_{int} is applicable when the current time index t verifies $AFTER(t, I_d)$

So, when the system is in $s \in S$ and is in a waiting loop, we have $t \in [t_{in}(s), t_{in}(s)+t_a(s)] = I_d$ and then τ_{ext} is applicable. When $t > t_{in}(s) + t_a(s)$, the state life-time is "over". Therefore, the internal transition function τ_{int} is triggered off. For effective management of this mechanism, Zeigler associates at each state, a variable e (for elapsed time), initialized at $t_a(s)$ when the system comes into a state, and is decreased proportionally by time spent in the state.

Having introduced the mechanism for the life-time notion, a loop on a state needs to be more specific. We can have either a waiting loop or a loop for a state reexecution. The waiting loop allows the time index to evolve in time without a state change. A loop for reexecution is, indeed, an external transition from a state to itself. So, for the latter loop, the elapsed time variable e associated with the considered state is initialized, as opposed to the waiting loop, where e decreases to model the evolution of the time. Having the notion of time evolution, a fact $F(p, I)$ in a state variable is verified at t when $t \in I$. However, to capture an event condition $e(p, i)$, the easiest way to recognize a sudden change is to "remember" the instant i', defined in definition 5.3, and verify OCCUR($\neg p, i'$)$\wedge$ OCCUR(p, i). But, in our representation, we do not keep any values from the past. To solve this problem, we must create another state variable for each state and input variable. This new variable can take the value RISE, FALL or STABLE. We can defined other values to fill our needs. These variables, called behavioral variables, are computed at each instant. So, for the input variables, the behavioral variables are up-dated every time a change occurs in them. When a state change caused by one of the transition functions occurs, an operation allows the computation of the behavioral variable values associated with all the state variables. However, if the system takes the waiting loop of a state then all the behavior variables related to state variables get the value STABLE. Because we represent the event notion in this fashion, we have a system uniformly modeled using the fact notion. So, an event condition $e(p, i)$ becomes a fact $F(p', I)$ as follows: $p' = p \wedge p'EVENT$ and $I = [i', i]$ where $EVENT = RISE \vee FALL$. Thus, all conditions can be checked the same way at a given instant.

The pseudo-state machine where time is encapsulated is a deterministic system. Therefore, the system under specification can be only in one state at the time. So, to prevent conflicts in the choice of the future state, the notion of priority is introduced. To treat these conflicts, we define a priority function Γ such as $\Gamma : S \times S \to \aleph$. The greatest priority transition is the one with the highest numeric value. To solve a conflict, only the transitions involved in the conflict are in concurrence. An illustration of the behavior of this function is as follows:

$\tau_{ext}(S_1, cond_1) = S_2;$

$\tau_{ext}(S_1, cond_2) = S_3;$

$\tau_{ext}(S_1, cond_3) = S_4;$

$\Gamma(S_1, S_2) = 1;$

$\Gamma(S_1, S_3) = 3;$

$\Gamma(S_1, S_4) = 2.$

The system is in state S_1. Suppose that at t, $cond_1$ and $cond_3$ hold. A conflict occurs:the system can go into either S_2 or S_4. The resolution of this conflict consists of comparing the transition priorities for S_2 and S_4. In our example, we have $\Gamma(S_1, S_4) > \Gamma(S_1, S_2)$. Therefore, the future state is S_4. Notice that only the priorities of the transitions in conflict have been considered.

5.2.2 Syntax and Semantic Using Ziegler Formalism

To define properly the syntax and the semantic of the proposed VHLLS process, a model needs to be defined. So, the model $M_{zeigler}$ of the pseudo-state diagram (introduced in the preambule of this chapter) inspired by the Zeigler theory is the following:

$$M_{zeigler} = < S, I, F, O, \tau_{int}, \tau_{ext}, \psi_z, t_a, \Gamma >$$

where

- S represents the node set of the model corresponding to a set of state sub-sets of the system;
- I represents the input variable set of the internal model corresponding to the input variable set of the real system;

- F represents the conditional transition of one node to another. The composition of these conditions are realized with the input and state variables. This corresponds in reality to a state change condition in the system;
- O represents the action set associated with nodes. These actions represent the active part associated with states of the system. These actions are written in VHDL. The operative part can only modify the values of the state, non-state and output variables. We can have no actions associated with a node. Notice that states "included" in the same node are characterized by the same operative part;
- τ_{int}: S $\rightarrow$ S is the internal transition function. It is in relationship with the life time of a node:
- τ_{ext}: S $\times$ F $\rightarrow$ S describes the conditional transitions;
- ψ_z: S $\rightarrow$ O associates, at each node, an operative part which is a list of actions;
- t_a : S $\rightarrow$ $TIME$ associates a life-time to each state:
- Γ: S $\times$ S $\rightarrow$ $\aleph^-$ allows only one possible transition.

In the rest of this section, we illustrate each notion this model implies in order to represent a pseudo-state diagram with time encapsulated. So, the following illustrates the syntax and semantics to represent all the notions introduced by $M_{zeigler}$.

5.2.2.1 Actions Associated With a System State

Using the model $M_{zeigler}$, we can associate each system state with either actions or nothing. This is expressed by the output function ψ_z. This function is defined for each state. For this initial study, actions are described in a sequential manner using VHDL in the sequential mode. So, the result of the function ψ_z links with each state $s \in S$ a sequence of activities the system needs to perform when it is in s. If the system is in a state $s \in S$ and no action is associated with S then the result at the request $\psi_z(s)$ is the empty set ($\emptyset$).

As an illustration of the output function ψ_z, let us consider the following:

1. A state $s \in S$ has a list of actions to perform, so ψ_z looks like:

$$\psi_z(s) = \left\{ \begin{array}{l} action_1; \\ action_1; \\ \vdots \end{array} \right\}$$

2. A state $s \in S$ does not have anything to perform in s, so ψ_z looks like:

$$\psi_z(s) = \emptyset$$

For this initial study, these two cases are the only form of descriptions the output function can have.

5.2.2.2 State Life-Time Function

This function is defined as $t_a : S \rightarrow TIME$. The TIME type has already been defined. Each state is associated to its life-time through this function. The range of possible values is in [0 ns; ∞ ns]. A life-time of 0 ns means the system needs to change state as soon as the actions associated with it have been completed. On the other hand, a time-life of ∞ ns means the system can stay in a given state indefinitely if no transition conditions apply.

As an illustration of the life-time function, several cases are presented. Let $s \in S$,

1. $t_a(s) = 20\ ns$ means that the system stays in s up to 20 ns;

2. $t_a(s) = 0$ *ns* means that the system changes state as soon as the actions associated with s have been completed;

3. $t_a(s) = \infty$ *ns* means that the system changes state only if a transition condition holds.

5.2.2.3 Transition Functions

There are different ways of expressing the transition from one system state to another. In fact, this depends on the nature of the transition. To take into consideration these different ways we defined, in section 5.2.1, the following three functions:

- τ_{int} is the internal transition function. This function is related to the life-time function. τ_{int} is automatically applied to produce a transition from a state s_1 to a state s_2 ($s_1, s_2 \in S$) when the time given by the function t_a, $t_a(s_1)$, has elapsed. In other words, t being the time index and s_1 a given system state, if $t \notin I_d(s_1)$ then $\tau_{int}(s_1) = s_2$.

- τ_{ext} is the external transition function. This function is in relationship with conditions made up of V_s and V_i. The condition is a parameter of the external transition function along with a state. These two parameters allow the computation of a future state as long as the time index is within $I_d(s)$. So, t being the time index and s_1 a given system state, if $t \in I_d(s_1)$ and *cond* (the transition condition) holds true then $\tau_{ext}(s_1, cond) = s_2$.

- Γ is the priority function. Because the pseudo-state diagram is deterministic, Γ allows decision making when the transition functions can compute more than one future state. So, each transition is weighted, enhancing its importance relative to the other ones, thus it may be in conflict. When conflicts occur, we check the priority using Γ and the next system state corresponds to the one with the highest priority. Let's say that we have $s_1, s_2, s_3 \in S$, and two possible transitions, defined as $\tau_{ext}(s_1, cond_2) = s_2$ and $\tau_{ext}(s_1, cond_3) = s_3$, if, at t, $cond_2$ and $cond_3$ hold true then we consult $\Gamma(s_1, s_2) < \Gamma(s_1, s_3)$. If this query is true then $\tau_{ext}(s_1, cond_3) = s_3$ is applied otherwise $\tau_{ext}(s_1, cond_2) = s_2$. By default, the internal transition has the lowest priority which means that if at the same instant, the life-time of the current state elapsed and a transition condition holds, then the external transition is applied prior to the internal one.

5.2.3 Interpretation of Conditions

This section gives an interpretation of notions defined in section 5.2.1 and shows the method of defining transition conditions.

5.2.3.1 Simple Conditions

For the model $M_{zeigler}$, a fact and an event are represented in the same way. To enhance the difference between them, we added a new variable to each state and input variable. This variable can have, for instance, three values: FALL, RISE and STABLE. This variable is viewed as a boolean attribute describing the state or input variable evolution. The syntax is to put a quote and the attribute name after a type or object instantiation name. One attribute which has a behavior similar to our behavioral variables is EVENT in VHDL. This attribute returns a boolean value. It returns a true value (at the instant and only for that instant) when a variable changes its value during a simulation cycle. In the model $M_{zeigler}$, an event is a combination of two facts. The first one considers the past of a variable and the second one, a given time. So when we want to capture an event such as the increase in value of a state variable or input variable (v), we interpret this event as

$v'EVENT\ and\ v(i') < v(i)$ ($v(i)$ represents the value of v at the time index i) e.g. $v'RISE$ becomes $v'EVENT\ and\ v = 1$ where v is a boolean variable. By the same token, when we want to capture an event such as the decrease in value of a state variable or input variable (v), we interpret this event as $v'EVENT\ and\ v(i') > v(i)$ e.g. $v'FALL$ becomes $v'EVENT\ and\ v = 0$ where v is a boolean variable. It becomes obvious that, for a fact condition, the state or input variables which constitute the condition must be stable.

5.2.3.2 Composed Conditions

For each operator defined in section 5.2.1, each term must obey the rules described in section 5.2.2.1. Three operators were defined:

- COMPADD:
- COMPMULT:
- PIPE.

In the model $M_{zeigler}$, the composition of conditions is viewed as a single condition from the standpoint of the external transition function. Therefore, when we have:

$$\tau_{ext}(s_1, cond) = s_2$$

, where the condition *cond* can be expressed as:

- $cond = cond_1\ COMPADD\ cond_2$ which means that the condition transition is performed when either $cond_1$ or $cond_2$ holds under life-time constraints. Formally, the COMPADD operator is defined in section 5.2;
- $cond = cond_1\ COMPMULT\ cond_2$ which means that the condition transition is

performed when $cond_1$ and $cond_2$ hold at the same time under life-time constraints. Formally, the COMPADD operator is defined in section 5.2;

In contrast to the above operators, the PIPE operator defines a sequence of two conditions such as $cond = cond_1 \ PIPE \ cond_2$. This means that to have $cond$ holding, $cond_1$ must hold true first and then $cond_2$ is evaluated. Conceptually, this operator can be viewed as a combination of two τ_{ext} with a virtual state between the initial state and the final state. So, when we have:

$$\tau_{ext}(s_1, cond_1 \ PIPE \ cond_2) = s_2$$

, we rewrite this transition function as follows:

Let s_1^v being a virtual state of the system,

$$\tau_{ext}(s_1, cond_1) = s_1^v$$

$$\tau_{ext}(s_1^v, cond_2) = s_2$$

The properties of the new virtual state are:

- $t_a(s_1^v) = t_a(s_1) - e$ where e is the elapsed time of being in s_1
- $\tau_{int}(s_1^v) = \tau_{int}(s_1)$

5.2.4 Example of the Zeigler Model

This example uses the specification of the RAM introduced in Chap. 2. This system is composed of seven states: $INIT0, INIT1, WAIT, R/W, R, W, ERR$. For each state, we define the transition functions, the output function, the transition priority and their life-time. So for $INIT0$, we have:

- $\tau_{ext}(INIT0, "NSRT =' 0'") = INIT1$
- $\tau_{int}(INIT0) = \emptyset$
- $t_a(INIT0) = \infty$
- $\Gamma(INIT0, INIT1) = 0$

For $INIT1$, we have:

- $\tau_{ext}(INIT1, "NSRT =' 1'") = WAIT$
- $\tau_{int}(INIT1) = \emptyset$
- $t_a(INIT1) = \infty$
- $\Gamma(INIT1, WAIT) = 0$

For $WAIT$, we have:

- $\tau_{ext}(WAIT, "CS =' 1'") = R/W$
- $\tau_{ext}(WAIT, "NRST =' 0'") = INIT1$
- $\tau_{int}(WAIT) = \emptyset$
- $t_a(WAIT) = \infty$
- $\Gamma(WAIT, R/W) = 1$
- $\Gamma(WAIT, INIT1) = 0$

For R/W, we have:

- $\tau_{ext}(R/W, "(RD =' 1')\ COMPMULT\ (WR =' 0')") = R$
- $\tau_{ext}(R/W, "(RD =' 0')\ COMPMULT\ (WR =' 1')") = W$

- $\tau_{ext}(R/W,$ "$((RD =' 1')\ COMPMULT\ (WR =' 1'))COMPADD\ ((RD =' 0')\ COMPMULT\ (WR =' 0'))$") $= ERR$
- $\tau_{int}(R/W) = \emptyset$
- $t_a(R/W) = \infty$
- $\Gamma(R/W, W) = 0$
- $\Gamma(R/W, R) = 1$
- $\Gamma(R/W, ERR) = 2$

For R, we have:

- $\tau_{ext}(R,$ "$NRST =' 0'$"$) = INIT1$
- $\tau_{int}(R) = WAIT$
- $t_a(R) = 1ns$
- $\Gamma(R, WAIT) = 0$
- $\Gamma(R, INIT1) = 1$

For W, we have:

- $\tau_{ext}(W,$ "$NRST =' 0'$"$) = INIT1$
- $\tau_{int}(W) = WAIT$
- $t_a(W) = 1ns$
- $\Gamma(W, WAIT) = 0$

- $\Gamma(W, INIT1) = 1$

For ERR, we have:

- $\tau_{int}(ERR) = INIT0$
- $t_a(ERR) = 1ns$
- $\Gamma(ERR, WAIT) = 0$

5.3 Intelligence Built-In VHLLS Design Methodology

This section presents an evolution of the previous VHLLS model toward an "intelligence" built-in model. This model is built on a knowledge base which gives the VHLLS model extra features such as reasoning ability and greater flexibility with the specification description to capture. So, the first part of this section reviews a few knowledge based methods. Of these methods, one has been identified as more appropriate to our problem and is applied for the VHLLS model.

5.3.1 Knowledge Representation

Among knowledge based methods, three have been selected for their main characteristics. These three methods are: Rough Sets, InfoSchemata and Conceptual Graphs. The rough set method can optimize the amount of knowledge needed to describe a universe. The InfoSchemata method has the very useful ability of organizing knowledge in an abstract manner. Indeed, the knowledge is classified in a few levels of abstraction and a relationship between each of them creates the meaning of the knowledge base. Finally, the conceptual graph method has built-in operators that allow the knowledge base to evolve, expand and manipulate knowledge. Note that all the following sections which review methods to represent knowledge are self contained regarding symbols and notations.

5.3.1.1 Overview Of the Rough Sets Theory

In this section, an overview of the concept of rough set theory is presented. Thereafter, a simple example illustrates the whole idea of this method.

Rough set theory [PAW91] defines a universe of objects (U). In this universe, there exist relationships between objects. Each relationship classifies these objects into families. Having classified objects into subsets according to relations R (families), a knowledge base can be defined. This base is given by K = (U, R') where R' is a family of equivalence relations over U.

The goal of this theory is to classify and manipulate knowledge in a universe. To do so, notions and relations are defined. When a set of objects included in U is given, the goal is to know what represents this set in the universe and to associate its families in order to characterize it. Basic sets are defined as having the following approximations of sets : R-lower approximation of a set X (set of objects classified without ambiguity as elements of X), R-upper approximation of a set X (set of objects possibly classified as elements of X), R-boundary of X ({R-upper}∩{R-lower}), R-positive region of X (equal to {R-lower}), R-negative region of X (U-{R-upper}). Properties are derived from these notions. For practical utilization of this theory, data tables are constructed. Some operations can be applied to this representation, such as a reduction of attributes in data tables. In the process of model building, it should be possible to identify and eliminate redundant attributes without losing any essential information. Another functionality is decision rules. Non-redundant descriptions characterize potentially important patterns in data. The patterns are expressed as decision rules linking the presence, or absence, of specific conditions (attributes) with an outcome.

To illustrate the Rough set theory, the example which follows presents the manipulation and optimization of a knowledge base. Assume the following decision table:

U	a	b	c	d	e
1	1	0	0	1	1
2	1	0	0	0	1
3	0	0	0	0	0
4	1	1	0	1	0
5	1	1	0	2	2
6	2	1	0	2	2
7	2	2	2	2	2

where *a*, *b*, *c* and *d* are condition attributes (input) and *e* is a decision attribute (output). The attribute *c* appears to be dispensable so the column *c* can be removed. The next step is the computation of the core value of that decision table:

U	a	b	d	e
1	-	0	-	1
2	1	-	-	1
3	0	-	-	0
4	-	1	1	0
5	-	-	2	2
6	-	-	-	2
7	-	-	-	2

where '-' means 'do not care'. Therefore by assigning a proper value to these '-', the resulting table is:

U	a	b	d	e
1	1	0	×	1
2	1	0	×	1
3	0	×	×	0
4	×	1	1	0
5	×	×	2	2
6	×	×	2	2
7	×	×	2	2

Because decision rules 1 and 2 are identical, and so are rules 5, 6 and 7, the final table is:

U	a	b	d	e
1,2	1	0	×	1
3	0	×	×	0
4	×	1	1	0
5,6,7	×	×	2	2

This solution is referred to as *minimal*. Therefore, this method would be useful for classifying a knowledge base described using either InfoSchemata or Conceptual graph. That, in turn, would allow the use of another knowledge base optimization method which retains the meanings of both methods.

5.3.1.2 Overview of the InfoSchemata Theory

InfoSchemata [JM94] defines a methodology for representing and developing knowledge bases. As shown in Fig. 5-5, InfoSyntax, InfoSchema (abstractions at the general level) and

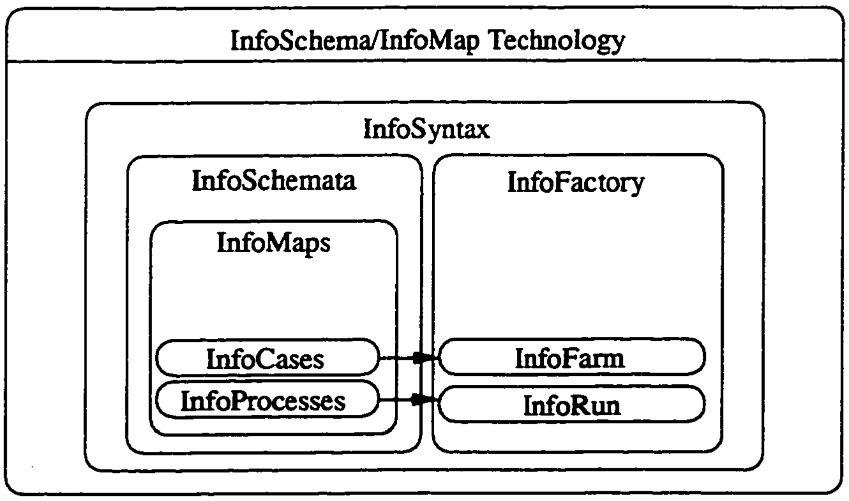


Figure 5-5: InfoSchema/InfoMap Structure

InfoFactory compose a framework to capture and manipulate knowledge.

InfoSyntax defines the syntax used to model concepts within the InfoSchema/InfoMap methodology. Two levels of abstractions characterize that method: the InfoSchema level and the InfoMap level.

In the InfoSchemata approach ("pattern" and "schemata" are synonymous terms), a vocabulary [JC91] can be used to derive schemata. These schemata are described in terms of sets and the relationships between them. The general format of the universal schema is given in the following:

```
Universal schema ::= [ [A]   {set_name}
                       [Y]   {set_name}
                       <Z>   {set_name}
                       (W)   {set_name}
                       <U>   {set_name} ]
```

Within the InfoSyntax, a hierarchy of relationships is defined allowing a mapping between sets which assigns specific set roles (Tab. 5.4): A, Y, Z, W, and U.

A::=	partition ;
Y::=	K - identifier; O - identity;
	H - hierarchy; I - generalization;
	P - aggregation;
Z::=	X - qualifier; M - association;
	F - flow; G - guard or goal;
	S - sequence; V - value or instance;
W::=	L - sequential state transitions;
	C - concurrent state transitions;
U ::=	User defined

Table 5.4: Set Roles

An InfoMap is created by enumerating sets and populating relationships defined by the InfoSchemata. This instantiation process implies that set roles are also instantiated to specify a role allocated to a role member (Tab.5.5).

Legal Set Role	Legal Set Element Role
A	v-column marker
K	id-unique identifier
O	o-column marker
H	h-root tree; 1..n-part marker
P	w-whole; c-part; v-visible; h-hidden part
I	p-parent; c-child
X	x-qualifier marker
M	v-row market; k-key attribute
F	u-used input; o-produced output
G	t-true; f-false; T-implied true; F-implied false
S	1..n-position in sequence, integer
V	instance, value, string
L	s-source; d-destination; l-loop; a-assertion; e-exemption
C	c-concurrent

Table 5.5: Member Roles

Set roles and associated set member roles are the core of InfoSchemata and InfoMap notation.

Therefore, using these mappings between concepts, knowledge can be synthesized. The manipulation of this knowledge can be realized through an InfoProcess which examines, partitions, and merges knowledge.

5.3.1.3 Overview of the Conceptual Graphs Theory

Conceptual graphs were introduced by Sowa in 1984 [SOW84] to represent and manipulate knowledge. In the process of capturing knowledge, a particular notion of perception is crucial. This notion allows the creation of a *working model* that represents and interprets sensory input. Two components constitute this working model, they are: a sensory input composed of a mosaic of percepts and a *conceptual graph* to fit percepts together. The basic goals of perception mechanisms are:

- to generate *sensory icons* to capture external stimulations;
- to compare these icons with percept to see if they match, called *associative comparator*;
- to generate a close approximation of the input and build a conceptual graph to store it, called *assembler*;
- conceptual mechanisms process *concrete concepts* that have associated percept and *abstract concepts* that do not have any associated percept.

The process of perception generates a structure u called a *conceptual graph* in response to some external entity or scene e:

- the entity e gives rise to a sensory icon s;
- the associative comparator finds one or more percept $p_1, p_2, \ldots, p_n$ that matches all or parts of s;
- the assembler combines the percept $p_1, p_2, \ldots, p_n$ to form a working model that approximates s;
- if such a working model can be constructed, the entity e is said to be recognized by the percept $p_1, p_2, \ldots, p_n$;
- for each percept p_i in the working model, there is a concept c_i called the interpretation of p_i;
- the concepts $c_1, c_2, \ldots, c_n$ are linked by conceptual relations to form the conceptual graph u.

Conceptual relations specify the *role* that each percept plays : one percept may match a part of an icon to the right or left of another percept. A representation and interpretation of a conceptual graph are:

- a linear form: [Conceptl] → (Rel) → [Concept2];
- a graphical form similar to the linear form is illustrated in Fig.5-6.

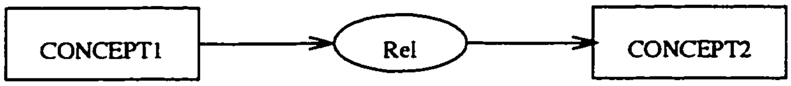


Figure 5-6: Basic Conceptual Graph: Graphical Representation

having the meaning: the *Rel* of a *Conceptl* is a *Concept2* e.g. with the following interpretation: "the **Instrument** of **APPLY** concept is **DEVICE** concept", the corresponding conceptual graph is shown in Fig.5-7.

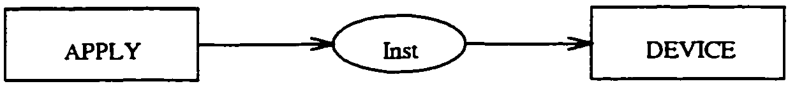


Figure 5-7: Inst of APPLY is DEVICE: a Conceptual Graph Representation

A conceptual graph is a finite, connected, bipartite graph. The two kinds of nodes of the bipartite graph are *concepts* and *conceptual relations*:

- Concept nodes: represent any entity, action or state that can be described in languages. For an AI standpoint, this kind of node encodes information in networks or graphs : concepts are a basic unit for representing knowledge;

- Conceptual relation nodes: show the roles that each entity plays. In other words, these nodes show how the concepts are interconnected.

As an illustration, two examples of concept nodes are given from [CYR94]:

- [DEVICE] embraces all hardware elements;
- [VALUE] covers the notions of data and message as well as software (commands and programs).

Two examples of conceptual relation nodes are given from the general conceptual graph theory:

- (Inst) links an [ENTITY] to an [ACT] in which the entity is causally involved;
- (Dur) links a [STATE] to a [TIME-PERIOD], during which the state persists.

Every conceptual relation has one or more arcs, each of which must be linked to some concept. If a relation has n arcs, it is said to be n-*adic* and its arcs are labeled 1, 2, ... n. The term *monadic* is synonymous with 1-adic, *dyadic* with 2-adic, and *triadic* with 3-adic. A single concept by itself may form a conceptual graph, but every arc of every conceptual relation must be linked to some concept. To be consistent, some assumptions are necessary such as (i) concepts are discrete units, (ii) combinations of concepts are not diffuse mixtures, but ordered structures, and (iii) only discrete relationships are recorded in concepts. Continuous forms must be approximated by patterns of discrete units.

A conceptual graph has no meaning in isolation. Only through the semantic network, concepts and relations which link context, language, emotion and perception, make sense. A conceptual graph can be displayed using two representations. The first one is in a linear form with:

- [...] representing a concept;
- (...) representing a conceptual relation.

Some concept or relation must be the head of this representation. A variable is noted as *x. Relations connected to the concept head are listed on subsequent lines after the symbol "-". The end of a graph is signaled by a period ".". Finally, a comma "," represents the end of subsequent lines. The second representation is graphical where a *square* is a concept, a *circle* a conceptual relation and an *arrow* puts in place the relations between nodes.

These possible representations of a conceptual graph are illustrated as follows:

- Linear representation:

  [ACTION : is reset] -

  $\rightarrow$ (Agnt : by) $\rightarrow$ [EVENT : # interruption],

  $\rightarrow$ (Obj) $\rightarrow$ [COUNTER : #timer].

- Graphical representation (see Fig.5-8).

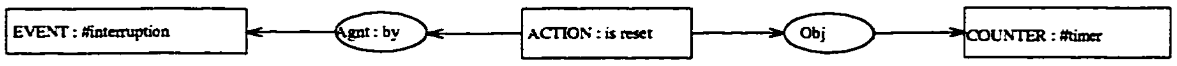


Figure 5-8: Example of Semantic Network in Conceptual Graphs

The notion of *type* in conceptual graphs is a classic one and means family resemblance. One type principle reads: "the logical type or category to which a concept belongs is the set of ways in which it is logically legitimate to operate with it" [RYL49]. Here we introduce the function *type* which maps concepts into a set T, whose elements are called type labels.

Concepts c and d are of the same type if $type(c) = type(d)$. As an illustration, let a concept c, $c =$ [Type Label] $\Leftrightarrow type(c) =$ Type Label.

In conceptual graphs, different meanings for knowledge or concepts can be classified to express generality or instantiation of an concept. Two kinds of markers can be identified: individual and generic markers. An analogy can be drawn with nouns in natural language where individual markers are like determinate nouns (example: the city) especially proper nouns (example Durham) in that they designate a specific object, whereas generic markers can be seen as indeterminate nouns (example: a city) which designate a class of objects with a similar set of characteristics (buildings, streets, ...). In conceptual graphs, an individual marker is specified by an identifier like [Type Label : ident] and a generic marker by an asterix like [Type Label : *] or simply [Type Label]. Here we introduce a function *referent* which corresponds to the identifier of a concept (ex: referent(Type Label) = ident).

Individual concepts correspond to constants in logic and programming languages, and generic concepts correspond to variables. In fact, variables like *x or *y in the linear notation are simply the generic marker *, followed by an identifier to indicate cross references e.g.:

[COUNTER : #timer]: in the concept "counter", we consider the "timer" to be a type counter of which referent(COUNTER) = #timer.

[COUNTER : *t]: in the concept "counter", we consider a counter.

A formula operator Φ is introduced which translates a conceptual graph into a logical formula. The operator Φ maps conceptual graphs into formulae in first-order predicate calculus. If u is any conceptual graph, then Φu is a formula determined by the following construction:

- if u contains k generic concepts, then assign a distinct variable symbol $x_1, x_2, \ldots, x_k$ to each one;

- for each concept c of u, let *identifier(c)* be the variable assigned to c if c is generic, or *referent(c)* if c is individual;
- each concept c represented as a monadic predicate whose name is the same as *type(c)* and whose argument is *identifier(c)*;
- each n-adic conceptual relation r of u represented as an n-adic predicate whose name is the same as *type(r)*. For each i from 1 to n, let the ith argument of the predicate be the identifier of the concept linked to the ith arc of r.
- then Φu has a *quantifier prefix* $\exists x_1 \exists x_2 \ldots \exists x_k$ and a *body* consisting of the conjunction of all the predicates for the concepts and conceptual relations of u.

Therefore, if a conceptual graph is as follows:

u = [RESET] -

$\rightarrow$ (Agnt) $\rightarrow$ [EVENT]

$\rightarrow$ (Obj) $\rightarrow$ [MEMORY]

$\rightarrow$ (Nval) $\rightarrow$ [VALUE].

Then the resulting Φu is:

$\Phi u = \exists x, y, z, w[RESET(x) \wedge Agnt(x, y) \wedge EVENT(y) \wedge Obj(x, z) \wedge MEMORY(z) \wedge Nval(x, w) \wedge VALUE(w)]$

The notion of canonical graphs is defined to distinguish the meaningful graphs that represent real or possible situations in the external world. Certain conceptual graphs are *canonical*. New graphs may become canonical or be "canonized" by any of the following three processes:

- perception: any conceptual graph constructed by the assembler in matching a sensory icon is canonical;

- formation rules: new canonical graphs may be derived from other canonical graphs by means of the rules *copy, restrict, join, and simplify*;
- insight: arbitrary conceptual graphs may be assumed to be canonical.

In a knowledge-based system, insight corresponds to the introduction of new graphs by a knowledge engineer who encodes information more efficiently. The formation rules are a generative grammar for conceptual structures. All deductions and computations on conceptual graphs involve some combination of these rules.

These formation rules are described in the following list. Let u and v be conceptual graphs, w derives from them, then the formation rules are:

- *copy* rule: an exact copy of a canonical graph is also a canonical graph, $w = u$;
- *restrict* rule: replace the type label of a concept with the label of a subtype. This rule may also convert a generic concept into an individual concept. For any concept c in u, *type(c)* may be replaced by a subtype : if c is generic, its referent may be changed to an individual marker. These changes are permitted only if *referent(c)* conforms to *type(c)* before and after the change;
- *join* rule: merge identical concepts. If a concept c in u is identical to a concept d in v, then let w be the graph obtained by deleting d and linking to c all arcs of conceptual relations that had been linked to d;
- *simplification* rule: if conceptual relations r and s in the graph u are duplicates, then one of them may be deleted from u together with all its arcs.

5.3.1.4 Suitable Knowledge Representation

The knowledge bases, previously presented, specific strengths are:

- Optimization capabilities of the knowledge base in the rough set methodology;
- Good organization of the knowledge emphasizing a hierarchy and partition of the knowledge in the InfoSchemata approach. It is very convenient when someone has to create and manipulate a knowledge base;
- Good structure of the knowledge base with an enhancement of the evolution of the knowledge base in the conceptual graphs methodology.

For the purpose of the VHLLS synthesis process, the most important criterion to consider is the ability to improve and increase the potential of this process without redefining the whole process. So, for that matter, the conceptual graph approach is the most suitable method for our research problem.

5.3.2 Syntax and Semantics Using Conceptual Graphs Formalism

In the pseudo-state graph, encapsulated time is a very important notion. Therefore, to build a conceptual graph, notions coming from state diagrams and time must be captured. When a state graph is analyzed, some concepts come out such as transitions from one state to another. To make sure that a state graph behaves in a deterministic way, one relation seems critical: priority between two transitions.

The purpose of using conceptual graphs is to build a knowledge base capturing all the information necessary to generate a VHDL description. Another interest is that conceptual graph methodology is a representation flexible enough to be used to describe other specification models using a unique methodology. The main objective for a global environment is to provide specification descriptions which designers are looking for, such as state diagrams, petri nets, timing diagrams and so on. An embryo of this global environment is introduced in chapter 6 and we call it Specification Procedure for Electronic Circuits in Automation Language (SPECIAL). Furthermore, conceptual graphs can expand the spectrum of this knowledge base further. Conceptual graphs offer a way to grapple with the information within the knowledge base and go further with specification methods like those using natural language as a communication vector. All these options are objectives and a direction to follow. Currently, the problem is to set up a knowledge base using conceptual graphs and to find a way of generating a behavioral description in VHDL. Therefore, the first step is to define canonical graphs, conceptual types, and conceptual relations for this problem.

The first canonical conceptual graph to be defined is the TRANSITION concept which captures a transition from a beginning state to an ending state. A transition is controlled by a condition. The canonical conceptual graph is the following:

[TRANSITION] -

→ (Beginning) → [STATE],

→ (Ending) → [STATE],

→ (Inst) → [PROPOSITION].

In this graph, conceptual relations are defined as follows:

- Beginning specifies the state from which the transition leaves;
- ending specifies to what state the transition goes;
- Inst, for instrument, links a transition to a proposition which controls the state change.

In this conceptual graph, the instrument of a transition is a PROPOSITION concept which is a type of symbolic information. To construct this proposition, some time constraint relations must be introduced. This corresponds, in fact, to a way of encapsulating the time in our pseudo-state graph. Therefore, the relations defined by Allen [ALL84] (introduced in Section 5.2) and introduced in a conceptual graph by Cyre [CYR94] are:

- Meets: relates two intervals I_1 and I_2. The idea of this interval relation is that the interval I_1 finishes when the other one I_2 starts:

  [INTERVAL] → (Meet) → [INTERVAL]:

- Overlaps: relates two intervals I_1 and I_2. In this relation, I_1 has to start before I_2 and they overlap:

  [INTERVAL] → (Overlaps) → [INTERVAL];

- In: relates two intervals I_1 and I_2:

  [INTERVAL] → (In) → [INTERVAL].

  This relation is in fact a union of several interval relations. However, it is very convenient to define this relation as summarizing the situation in which one interval is wholly contained in another. Then In(I_1, I_2) is equivalent to:

  $$\text{During}(I_1, I_2) \vee \text{Starts}(I_1, I_2) \vee \text{Finishes}(I_1, I_2).$$

Other relations are introduced. They are specific to the pseudo-state diagram model:

- Fact: relates an interval I to a proposition p. The proposition p is a logic proposition independent of time. The interval I represents the time interval when p is verified:

  [INTERVAL] → (Fact) → [PROPOSITION]

This can be interpreted as: the fact in an interval is a proposition. More formally, a fact F(p, I) as defined in definition 5.2 is: let p be a proposition and $I \in \Upsilon$ such as F(p, I) $\Rightarrow \forall I', I" \in \Upsilon$:

- $BEFORE(I', I) \wedge BEFORE(I, I")\wedge$
- $HOLD(\neg p, I')\wedge$
- $HOLD(\neg p, I")$.

• Event: relates an instant i to a proposition p. The proposition p is a logic proposition independent of time. The instant i represents the moment when p is verified:

[INSTANT] → (Event) → [PROPOSITION]

where the INSTANT concept is a subconcept of INTERVAL. This can be translated as: the event at an instant is a proposition. More formally, an event e(p, i), as defined in definition 5.3 is: let p be a proposition and $i \in \Xi$ such as e(p, i) $\Rightarrow \forall i', i" \in \Xi$:

- $BEFORE(i', i) \wedge BEFORE(i, i")\wedge$
- OCCUR(¬p, i') ∧
- OCCUR(¬p, i").

So, to represent "NRST = '0' " as a fact, the corresponding conceptual graph representation is as follows:

[INTERVAL :#I] → (Fact) → [PROPOSITION :#"NRST = '0' "]

To represent NRST'RISE as an event, the corresponding conceptual graph representation is as follows:

[INSTANT :#i] → (Event) → [PROPOSITION :#"NRST'RISE"]

Having presented the concept of TRANSITION from one state to another one, it seems important to introduce the concept of STATE:

[STATE] -

→ (Link) → [TYPE]

→ (Name) → [WORD].

At this point, this concept is defined using an identification name relation which links a STATE concept to a WORD concept, naming the state. Another relation which is the Link relation is defined allowing a linkage between a STATE concept and a TYPE concept. The concept TYPE identifies whether the state is associated with an action or not, the "action", for now, being defined as a sequence of actions. Future improvements may have as one objective: to define a hierarchical structure allowing the introduction of other kinds of actions associated with a state (example: concurrence).

Therefore, when a state called "INIT0" having no actions associated with it, the corresponding conceptual graph model is as follows:

[1 : STATE] -

→ (Link) → [TYPE : #none]

→ (Name) → [WORD : #INIT0].

An important requirement for a state diagram is to guaranty the sequentiality of this representation. This requirement is addressed by the conceptual relation $<_{prior}$ between two transitions:

$$[\text{TRANSITION}] \rightarrow <_{prior} \rightarrow [\text{TRANSITION}].$$

This relation orders transitions when there is a need. When more than one transition leaves the same state, then a process has to choose which transition the system takes during a transition conflict. $<_{prior}$ links [TRANSITION: *x] to [TRANSITION: *y] where the transition *x has a higher priority than *y e.g let the transition #1 have the highest priority compare to the transition #2, the corresponding conceptual graph representation is:

$$[\text{TRANSITION: \#1}] \rightarrow <_{prior} \rightarrow [\text{TRANSITION: \#2}]$$

5.3.3 Example of the Conceptual Graphs Model

To illustrate this knowledge based approach, the case study defined in Section 2.3 is used to build the knowledge base of the RAM cell. First, the instantiation of the STATE concept is performed. It follows the definition of the transition between each state with the expression of the condition for a transition. So, the concept TRANSITION is instantiated for each transition and the concept PROPOSITION contains the condition of transition having time encapsulated. Finally, the priority relations between TRANSITION concepts are specified.

```
[1 : STATE ] -
        → (Link) → [TYPE : #none]
        → (Name) → [WORD : #INIT0].
[2 : STATE] -
        → (Link) → [TYPE : #none ]
        → (Name) → [WORD : #INIT1].
[3 : STATE] -
        → (Link) → [TYPE : #none ]
        → (Name) → [WORD : #WAIT].
[4 : STATE] -
        → (Link) → [TYPE : #none ]
        → (Name) → [WORD : #R/W].
```

```
[5 : STATE] -
            → (Link) → [TYPE : #action]
            → (Name) → [WORD : #R].
[6 : STATE] -
            → (Link) → [TYPE : #action ]
            → (Name) → [WORD : #W].
[7 : STATE] -
            → (Link) → [TYPE : #action ]
            → (Name) → [WORD : #ERR].
[8 : TRANSITION : *t_1] -
            → (Beginning) → [1 ]
            → (Ending) → [2]
            → (Inst) → [PROPOSITION: -
                          [INSTANT : * t_1] → (In) → [INTERVAL :{[0...t] | [t_12...t]}]
                          [INSTANT : *t ] → (>) → [INSTANT : * t_1]
                          [INTERVAL :#I] → (Fact) → [PROPOSITION :#"NRST = '0' "]
                          [INSTANT :  * t_1] → (In) → [INTERVAL : #I ].
                          ].
[9 : TRANSITION : *t_2] -
            → (Beginning) → [2 ]
            → (Ending) → [3]
            → (Inst) → [PROPOSITION: -
                          [INSTANT : *t_2 ] → (In) → [INTERVAL : *[t_1 ...t] ]
                          [INSTANT : *t ] → (>) → [INSTANT : *t_2 ]
                          [INTERVAL : # I] → (Fact) → [PROPOSITION : # "NRST = '1'  "]
                          [INSTANT : *t_2 ] → (In) → [INTERVAL : # I ].
                          ].
]
[10 : TRANSITION : *t_3] -
            → (Beginning) → [3 ]
            → (Ending) → [2]
            → (Inst) → [PROPOSITION: -
                          [INSTANT : *t_3 ] → (In) → [INTERVAL : *[t_2 ...t] ]
                          [INSTANT : *t ] → (>) → [INSTANT : *t_3 ]
                          [INTERVAL : # I] → (Fact) → [PROPOSITION : #"NRST = '0'  "]
                          [INSTANT : *t_3 ] → (In) → [INTERVAL : # I ].
                          ].
[11 : TRANSITION : *t_4] -
            → (Beginning) → [3 ]
            → (Ending) → [4]
            → (Inst) → [PROPOSITION: -
                          [INSTANT : *t_4 ] → (In) → [INTERVAL : *[t_2 ...t] ]
                          [INSTANT : *t ] → (>) → [INSTANT : *t_4 ]
                          [INTERVAL : # I] → (Fact) → [PROPOSITION : # "CS = '1'  "]
                          [INSTANT : *t_4 ] → (In) → [INTERVAL : # I ].
                          ].
```

```
[12 : TRANSITION : *t_5] -
          → (Beginning) → [4 ]
          → (Ending) → [7]
          → (Inst) → [PROPOSITION: -
                         [INSTANT : *t_5 ] → (In) → [INTERVAL : *[t_4 . . . t] ]
                         [INSTANT : *t ] → (>) → [INSTANT : *t_5 ]
                         [INTERVAL : # I] → (Fact) → [PROPOSITION :
                                                   #{"WR=RD='0' " | "WR=RD='1' "}]
                         [INSTANT : ∗ t_5] → (In) → [INTERVAL :#I ].
                         ].
[13 : TRANSITION : ∗ t_6] -
          → (Beginning) → [4 ]
          → (Ending) → [5]
          → (Inst) → [PROPOSITION: -
                  [INSTANT : ∗ t_6] → (In) → [INTERVAL : ∗ [t_4 . . . t]]
                  [INSTANT : *t ] → (>) → [INSTANT : ∗ t_6]
                  [INTERVAL : # I] → (Fact) → [PROPOSITION : # "WR='0' ∧ RD='1' "]
                  [INSTANT : ∗ t_6] → (In) → [INTERVAL : # I ].
                  ].
[14 : TRANSITION : *t_7] -
          → (Beginning) → [4 ]
          → (Ending) → [6]
          → (Inst) → [PROPOSITION: -
                  [INSTANT : *t_7 ] → (In) → [INTERVAL : *[t_4 . . . t] ]
                  [INSTANT : t ] → (>) → [INSTANT : ∗ t_7]
                  [INTERVAL : # I] → (Fact) → [PROPOSITION : # "WR='1' ∧ RD='1' "]
                  [INSTANT : ∗ t_7] → (In) → [INTERVAL : # I ].
                  ].
[15 : TRANSITION : *t_8] -
          → (Beginning) → [5 ]
          → (Ending) → [2]
          → (Inst) → [PROPOSITION: -
                         [INSTANT : ∗ t_8] → (In) → [INTERVAL : ∗ [t_6 . . . t]]
                         [INSTANT : *t ] → (>) → [INSTANT : ∗ t_8]
                         [INTERVAL : # I] → (Fact) → [PROPOSITION : # "NRST = '0'"]
                         [INSTANT : ∗ t_8] → (In) → [INTERVAL : # I ].
                         ].
[16 : TRANSITION : *t_9] -
          → (Beginning) → [6 ]
          → (Ending) → [2]
          → (Inst) → [PROPOSITION: -
                         [INSTANT : ∗ t_9] → (In) → [INTERVAL : ∗ [t_7 . . . t]]
                         [INSTANT : *t ] → (>) → [INSTANT : ∗ t_9]
                         [INTERVAL : # I] → (Fact) → [PROPOSITION : # "NRST = '0'"]
                         [INSTANT : ∗ t_9] → (In) → [INTERVAL : # I ].
                         ].
```

```
[17 : TRANSITION : *t_10] -
        → (Beginning) → [7 ]
        → (Ending) → [1]
        → (Inst) → [PROPOSITION: -
                    [INTERVAL : * [t5, t5 + 1ns]] → (Meet) → [INTERVAL : *I']
                    [INTERVAL : * [t5, t5 + 1ns]] → (Overlaps) → [INSTANT : * t10]
                    [INSTANT : * t10] → (Overlaps) → [INTERVAL : *I' ]
                    ].

[18 : TRANSITION : *t_11] -
        → (Beginning) → [5 ]
        → (Ending) → [3]
        → (Inst) → [PROPOSITION: -
                    [INTERVAL : * [t6, t6 + 1ns]] → (Meet) → [INTERVAL : *I']
                    [INTERVAL : * [t6, t6 + 1ns]] → (Overlaps) → [INSTANT : * t11]
                    [INSTANT : * t11] → (Overlaps) → [INTERVAL : *I' ]
                    ].

[19 : TRANSITION : * t12] -
        → (Beginning) → [6 ]
        → (Ending) → [3]
        → (Inst) → [PROPOSITION: -
                    [INTERVAL : * [t7, t7 + 1ns]] → (Meet) → [INTERVAL : *I']
                    [INTERVAL : * [t7, t7 + 1ns]] → (Overlaps) → [INSTANT : * t12]
                    [INSTANT : * t12] → (Overlaps) → [INTERVAL : *I' ]
                    ].

[10 ] → (> _prior) → [11]
[12] → (> _prior) → [13]
[13] → (> _prior) → [14]
[16] → (> _prior) → [18]
[17] → (> _prior) → [19]
```

5.4 Conclusion

As illustrated in Fig. 5-1, the main problem with the basic VHLLS approach[6] is that for each representation in the conceptual phase a specific translator has to be implemented to generate the intermediate description and another one to generate the behavioral description [VCSR94]. To avoid having a specific translation for each description model, the creation

[6]defined in the preamble of this chapter.

of a knowledge base can reduce the number of translations from the intermediate model to the behavioral description to one. This idea is also conducive for updating the knowledge base when a new representation is added to the description style set. Another advantage of using a knowledge base is that reasoning abilities are built into it. This approach can be applied, in turn, in such a way that it leads to the optimization of the device specification and by conducting this activity we have a preliminary optimization at the behavioral level. Finally, as shown in Fig. 5-1, the knowledge base can also be used to generate test vectors at the behavioral level, allowing for critical time-saving (it is not the purpose of this thesis to demonstrate the use of a knowledge base to generate test vectors: Further work is required). The automation of this task can replace either a manual approach or an Automatic Test Pattern Generator (ATPG) approach. At the behavioral level, the former one is unrealistic for the industry environment and the latter one is an NP-complex problem as shown in [SCG93].

Chapter 6

Specification Procedure for Electronic Circuits in Automation Language (SPECIAL)

This chapter illustrates the VHLLS from specification to a behavioral description in VHDL. First, VHDL is briefly introduced followed by the graphical interface used to model the pseudo-state diagram. Finally, the structure of the VHDL code is presented. This structure is a template of code.

6.1 Introduction to VHDL

This section presents VHDL fundamentals to help understand the translation of the VHLLS process. In this section, the subset of VHDL shown is only enough to understand the VHDL templates of section 6.2.3. The remaining notions of VHDL are very similar to ADA and can be found in the IEEE standard [IEE93].

6.1.1 BLOCK Statement

A block statement defines an internal block representing a portion of a design. This statement allows the use of concurrent statements in order to define interconnected blocks and processes that describe the overall behavior or structure of a design. Concurrent statements execute asynchronously with respect to each other.

```
concurrent-statement ::=
    block-statement
    | process-statement
    | concurrent-assertion-statement
    | concurrent-procedure-call
    | concurrent-signal-assignment-statement
    | component-instantiation-statement
    | generate-statement
```

Blocks may be hierarchically nested to support design decomposition. The blocks properties are :

- declaration encapsulation :
- hierarchy support.

The syntax of a block statement is the following :

```
block-statement ::=
      block-label :
      BLOCK [(guard-expression)]
            block-header
            block-declarative-part
      BEGIN
            block-statement-part
      END BLOCK [block-label] ;
block-header ::=
      [ generic-clause
      [ generic-map-aspect ;]]
      [ port-clause
```

```
        [ port-map-aspect ;]]
block-declarative-part ::=
        { block-declarative-item }
block-statement-part ::=
        { concurrent-statement }
```

6.1.2 PROCESS Statement

A process statement defines an independent sequential process representing the behavior of some portion of the design. This statement, in the same manner as the block statement, is a concurrent statement (seen above). The execution of a process statement consists of a repetitive execution of its sequence of statements. After the last statement in the sequence of statements is completed, the execution mechanism immediately continues with the first statement of the sequence of statements. A process statement is said to be a *passive process* if neither the process itself nor any procedure of which the process is a parent, contains a signal assignment statement. To control its execution, VHDL has an instruction named WAIT (see below) allowing a change to a passive process statement until an event on the sensitivity list of WAIT modifies the process statement to be active. The syntax of a process statement is the following :

```
process-statement ::=
        [process-label : ]
            PROCESS [(sensitivity-list)]
                    process-declarative-part
            BEGIN
                    process-statement-part
            END PROCESS[process-label]
process-declarative-part ::=
        { process-declarative-item }
process-declarative-item ::=
        subprogram-declaration
        | subprogram-body
        | type-declaration
        | subtype-declaration
```

| constant-declaration
| variable-declaration
| file-declaration
| alias-declaration
| attribute-declaration
| attribute-specification
| use-clause
process-statement-part ::=
{ sequential-statement }

6.1.3 WAIT Statement

The wait statement causes the suspension of a process statement or a procedure. The syntax of this statement is the following :

wait-statement ::=
WAIT [sensitivity-clause][condition-clause][timeout-clause] ;
sensitivity-clause ::= **ON** sensitivity-list
sensitivity-list ::= *signal*-name {, *signal*-name }
condition-clause ::= **UNTIL** condition
condition ::= *Boolean*-expression
timeout-clause ::= **FOR** *time*-expression

The sensitivity clause defines the *sensitivity set* of the wait statement. The execution of a wait statement causes the time expression to be evaluated to determine the *timeout interval.* It also causes the execution of the corresponding process statement to be suspended, where the corresponding process statement is the one that either contains the wait statement or is the parent of the procedure that contains the wait statement. The suspended process will resume, at the latest, immediately after the timeout interval has expired. The suspended process may also resume as a result of an event occurring on any signal in the sensitivity set of the wait statement. If such an event occurs, the condition in the condition clause is evaluated. If the value of the condition is TRUE, the process will resume. If the value of the condition is FALSE, the process will re-suspend. Such re-suspension does not involve the recalculation of the timeout interval.

6.2 SPECIAL

This section introduces the graphical language for modeling the pseudo-state diagram which encapsulates time.

6.2.1 Syntax

As defined in the chapter 1, the hypothesis restricts the domain of consideration to a pseudo-state diagram without hierarchy and actions associated with a state written in VHDL. Through these hypotheses, a specification language is defined.

The first step in a design flow is to define inputs and outputs by their relationships. When a designer wants to specify sequential circuits, the behavior of this system is usually transcribed with *nodes* and *arrows.* A node represents a state of the system in which either actions (written in VHDL), a graph type representation, or other representations to be defined (Petri net, timing diagram, ...) can be associated. Therefore, within the study's restriction, the system has only actions associated with a node. The second notion is an arrow, which represents the capability of changing state. The system behavior can be controlled by a condition associated with an arrow entailing a transition from one node to another. This type of arrow is called a *conditional arrow.* This representation is close to a state graph. Therefore, in order to refer to this last model, a node is designated as a *state* and an arrow represents a *conditional transition* from one state to another.

Therefore, the behavior of the system is controlled by these conditions which are sensitive to input variables of the system and variables computed in actions associated with a state. These actions can also contain variables:

- for intermediate computations, called computational variables. They can be "local" to a state or "global";
- to represent an output: output variables.

Consequently, four variables types are defined:

- input variables;
- local variables;
- global variables (they can be either variables intervening in the composition of conditions or global computational variables);
- output variables.

Thus, the composition of conditional transitions is performed with the input variables and the global variables. The other variable types (local and output) cannot be used. A conditional transition is expressed by means of two notions: fact and event.

6.2.2 Semantics

Once the notions and notations for the specification language have been defined, relations are specified. As we defined in Chapter 5, we have two categories of conditions: simple and composed conditions. This section defines the semantics for expressing these conditions with the specification language SPECIAL.

6.2.2.1 Simple Conditions

This section presents the simple conditions which are: facts, events and life-time.

6.2.2.1.1 Fact Relation: A fact is verified when the time interval defining the maximum duration in a given state, and the interval defining a fact are related as in Fig.6-1.

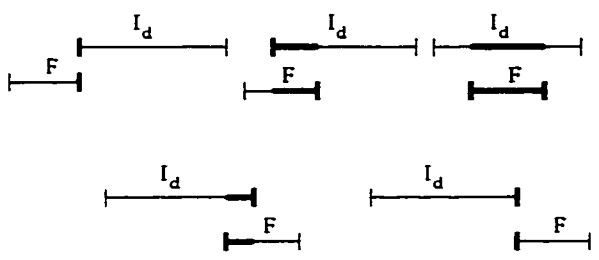


Figure 6-1: Timing Representation: Fact Verified

A fact is not verified elsewhere as illustrated in Fig. 6-2

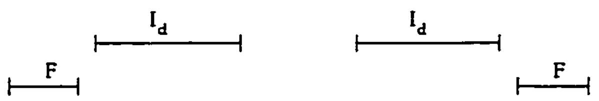


Figure 6-2: Timing Representation: Fact Not Verified

One generic case relating the user interface with the representation of a fact illustrated and interpreted is shown in Fig.6-3.

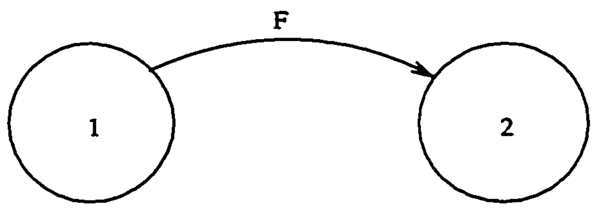


Figure 6-3: Graphical Representation of Fact

So, the transition condition from state 1 to state 2 (Fig.6-3) is a fact. In order to express it, we must write it as: F = "var = val" where var is an input variable or a non-computational variable and val is a value which affects var. If the system is in state 1 (as in the example) and fact F is true, the system will switch to state 2. If one of these two statements is not verified, this change of state will not be carried out. The following algorithm illustrates the representation behavior of a fact:

1. the operative part is computed ;
2. the computational variables are assigned their new value ;
3. the conditions of state change are consulted:
   (a) if the fact F is verified then
      - the time increases to t + δt ;
      - the non-computational global variables are assigned their new value ;
      - the state change is carried out ;
      - the system executes this principle in item - 1 - for the new state.

   (b) if the fact F is not verified then
      - the time increases to t + δt ;
      - the non-computational global variables are assigned their new value ;
      - the system stays in the same state ;
      - the system revalues the fact F with the same protocol as in - 3a - but without the revaluation of the non-computational variables.

6.2.2.1.2 Event Relation: An event is verified when the time interval defining the duration in a given state, and the instant of the event, are related as in Fig.6-4. An event is not verified elsewhere, as illustrated in Fig.6-5.

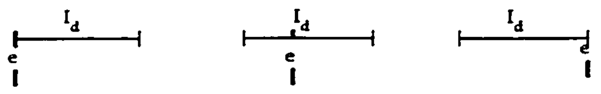


Figure 6-4: Timing Representation: Event Verified

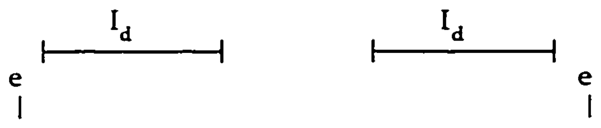


Figure 6-5: Timing Representation: event Not Verified

A generic case relating the user interface to the representation of a event is illustrated and interpreted as shown in Fig.6-6.

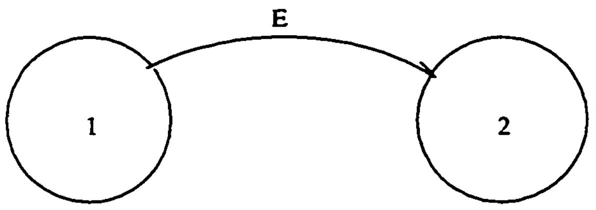


Figure 6-6: Graphical Representation of Event

So, the transition condition from state 1 to state 2 is an event. In order to express it, we must write it as: E = "up(var)" or " down(var)" where var is an input variable or a non-computational variable. If the system is in state 1 (in the example) and event E is true the system will switch to state 2. If one of these two statements is not verified, this change of state is not carried out. The following algorithm illustrates the representation behavior of an event:

1. the operative part is computed ;

2. the computational variables are assigned their new value ;

3. the conditions of state change are consulted:

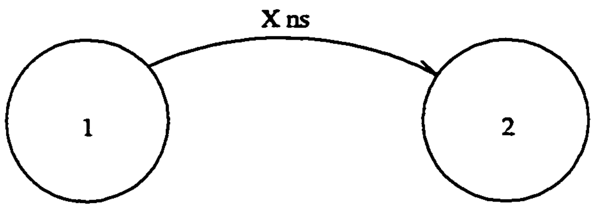


Figure 6-7: Life-Time Notion

(a) if the event E is verified then

- the time increases to t + δt :
- the non-computational global variables are assigned their new value ;
- the state change is carried out;
- the system executes this principle in item - 1 - for the new state.

(b) if the event E is not verified then

- the time increases to t + δt ;
- the non-computational global variables are assigned their new value ;
- the system stays in the same state ;
- the system revalues the event E with the same protocol as in - 3a - but without the revaluation of the non-computational variables.

6.2.2.1.3 Life-Time Relation: A generic case relating the user interface with the representation of a time condition is illustrated and interpreted in Fig.6-7. So, the transition condition from state 1 to state 2 is a time. In order to express it, we must write it as: T = "X unit" where X is an numerical value and unit is a time unit (ex: s, ms, μs, ns, ...). If the system is in state 1 (as in the example) and time T is over, the system will switch to state 2. If one of these two statements is not verified, this change of state is not carried out. The following algorithm illustrates the representation behavior of a time life notion:

1. the operative part is computed ;
2. the computational variables are assigned their new value ;
3. the conditions of state change are consulted:
   (a) if the time T is over then
   - the time increases to t + δt :
   - the non-computational global variables are assigned their new value :
   - the state change is carried out:
   - the system executes this principle in item - 1 - for the new state.

   (b) if the time T is not verified then
   - the time increases to t + δt ;
   - the non-computational global variables are assigned their new value ;
   - the system stays in the same state :
   - the system revalues the time T with the same protocol as in - 3a - but without the revaluation of the non-computational variables.

6.2.2.2 Composed Conditions

We have defined the notions of fact and event. Now, we attempt to associate these conditions, the result shall be called a composed condition (CC). We know that simple conditions are Boolean, so we can use operators from Boole's algebra. We suggest a few operators.

6.2.2.2.1 COMPADD Operator:

$$\text{CC} = SC_1 \text{ COMPADD } SC_2.$$

CC is true when SC_1 is true or SC_2 is true. Notice that this is similar to the OR operator, except that the time notion introduces constraints we must consider. We will see all the possibilities implied by this composition.

Note: The COMPADD operator is associative and commutative.

Let proposition p_1 and p_2 be respectively in SC_1 and SC_2. In our representation, we use the logic operator $\vee$ as OR from the classical logic OR. We characterize CC according to the different types SC_1 and SC_2.

- first case: $SC_1 = F(I_1, p_1)$ and $SC_2 = F(I_2, p_2)$ facts.

  CC = $F(I_1, p_1)$ COMPADD $F(I_2, p_2)$ is a fact $F(I', p')$ with:

$$\begin{cases} p' = p_1 \vee p_2 \\ I' = I_1 \cup I_2 \end{cases} ;$$

- second case: $SC_1 = e(t_1, p_1)$ and $SC_2 = e(t_2, p_2)$ events.

  CC = $e(t_1, p_1)$ COMPADD $e(t_2, p_2)$ is an event $e(t', p')$ with:

$$\begin{cases} p' = p_1 \vee p_2 \\ t' = t_1 \text{ or } t' = t_2 \end{cases} ;$$

- third case: $SC_1 = F(I, p_1)$ and $SC_2 = e(t, p_2)$ a fact and an event.

  CC = $F(I, p_1)$ COMPADD $e(t, p_2)$is an event or fact with:

$$\begin{cases} p' = p_1 \vee p_2 \\ R' = (I, t) \end{cases}$$

  and is expressed by the predicate $TRUE$ defined before: $TRUE(R', p')$.

We can represent this formalization by a graphical representation:

- CC is verified when one of the five cases in Fig. 6-8 holds;

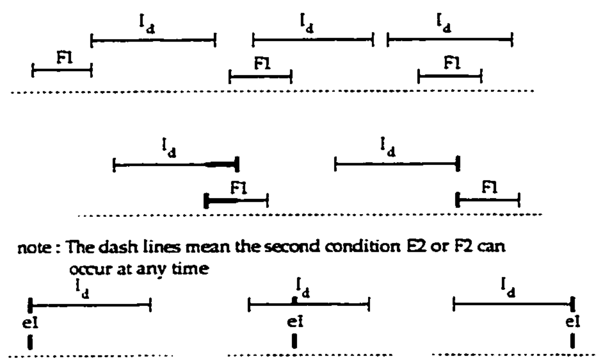

I_d
F1
note : The dash lines mean the second condition E2 or F2 can occur at any time
el

Figure 6-8: True Conditions of COMPADD Operator

- CC is not verified when, in for one of the possible combinations shown in Figs. 6-9, 6-10, 6-11, one of the cases holds.

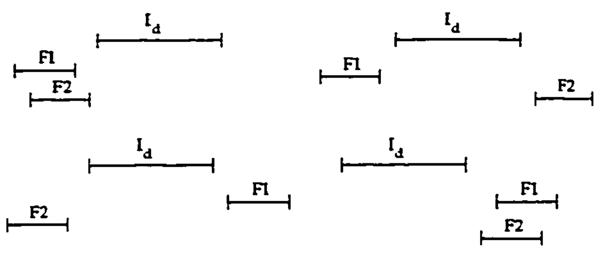


Figure 6-9: False Condition: F1 COMPADD F2

One Illustration of the behavior of COMPADD is proposed (its other possible combination can be found in Appendix A.1.1). For example, the condition for a transition is *CC = F1 COMPADD F2* and drawn as in Fig. 6-12.

The transition condition from state 1 to state 2 is: F1 COMPADD F2. F1 and F2 are expressed as explained above. If the system is in state 1 (as in the example) and the

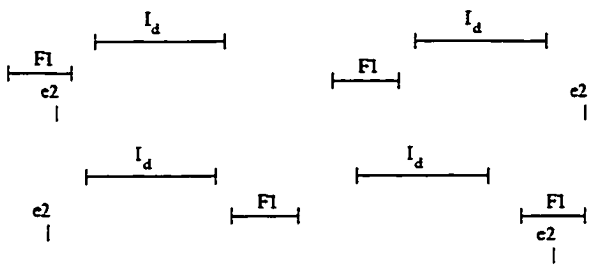


Figure 6-10: False Condition: F1 COMPADD e2

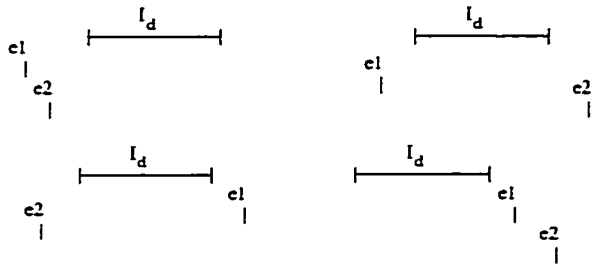


Figure 6-11: False Condition: e1 COMPADD e2

composed condition is true the system will switch to state 2. If one of these two statements is not verified, this change of state does not occur. The following algorithm illustrates the representation behavior of a composed condition using COMPADD:

1. the operative part is computed ;

2. the computational variables are assigned their new value ;

3. the conditions of state change are consulted:

    (a) if the condition F1 COMPADD F2 is verified then

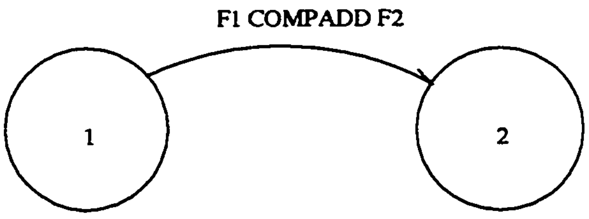


Figure 6-12: COMPADD With Two Facts

- the time increases to t + δt ;
- the non-computational global variables are assigned their new value ;
- the state change is carried out;
- the system executes this principle in - 1 - for the new state.

(b) if the condition F1 COMPADD F2 is not verified then

- the time increases to t + δt ;
- the non-computational global variables are assigned their new value :
- the system stays in the same state :
- the system revalues the condition with the same protocol as in - 3a - but without the revaluation of the non-computational variables.

6.2.2.2.2 COMPMULT Operator:

$$\text{CC} = SC_1 \text{ COMPMULT } SC_2.$$

CC is true when SC_1 is true and SC_2 is true. Notice that this is similar to the AND operator except that the time notion introduces constraints we must consider. We will see all the possibilities implied by this composition.

Note: The COMPMULT operator is associative and commutative.

Let propositions p_1 and p_2 be respectively in SC_1 and SC_2. In our representation, we use the logic operator $\wedge$ as AND from the classic logic AND. We characterize CC according to the different types SC_1 and SC_2.

- first case: $SC_1 = F(I_1, p_1)$ and $SC_2 = F(I_2, p_2)$ facts.

CC = $F(I_1, p_1)$ COMPMULT $F(I_2, p_2)$ is a fact $F(I',p')$ with:

$$\begin{cases} p' = p_1 \wedge p_2 \\ I' = I_1 \cap I_2 \\ \quad = [t_1, t_2] \end{cases};$$

- second case: $SC_1 = e(t_1, p_1)$ and $SC_2 = e(t_2, p_2)$ events.

  CC = $e(t_1, p_1)$ COMPMULT $e(t_2, p_2)$ is an event $e(t',p')$ with:

$$\begin{cases} p' = p_1 \wedge p_2 \\ t' = t_1 \ = \ t_2 \end{cases}:$$

- third case: $SC_1 = F(I,p_1)$ and $SC_2 = e(t,p_2)$ a fact and an event.

  CC = $F(I,p_1)$ COMPMULT $e(t,p_2)$ is an event or fact with:

$$\begin{cases} p' = p_1 \wedge p_2 \\ t' \in I \ and \ t' = t \end{cases}.$$

We can represent this formalization by a graphical representation:

- the two terms of COMPMULT are a fact, the composed condition is verified when the one of the cases shown in Fig. 6-13;

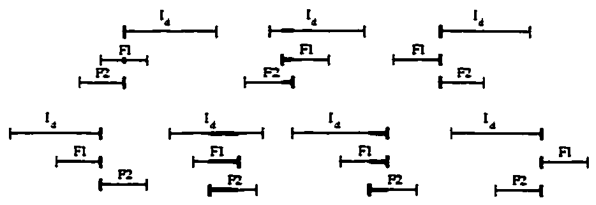


Figure 6-13: True Condition: F1 COMPMULT F2

- the two terms of COMPMULT are an event, the composed condition is verified when the one of the cases shown in Fig. 6-14;

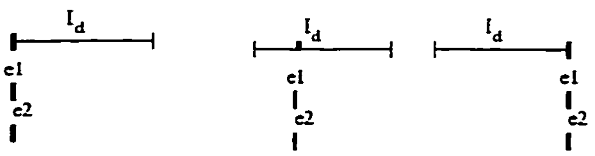


Figure 6-14: True Condition: e1 COMPMULT e2

- the two terms of COMPMULT are an event and a fact, the composed condition is verified when the one of the cases shown in Fig. 6-15:

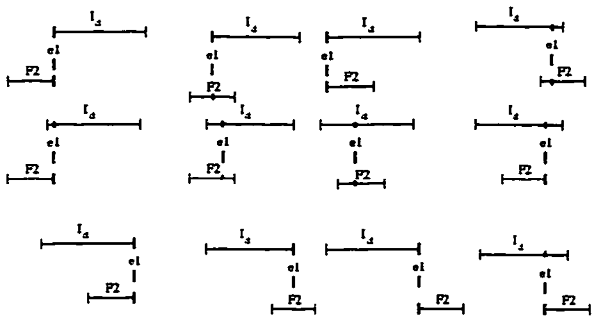


Figure 6-15: True Condition: e1 COMPMULT F2

All other possible cases express a composed condition which is false.

Now, it is shows the behavior of COMPMULT with one possible combination (the other possibilities can be found in Appendix A.1.2). For example, the condition for a transition is *cond = e1 COMPMULT e2* and drawn as in Fig. 6-16.

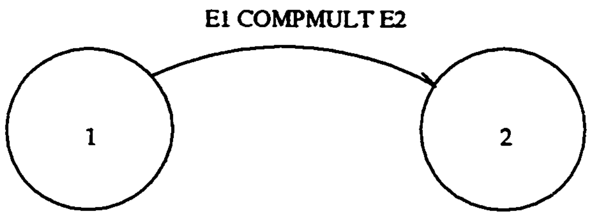


Figure 6-16: COMPMULT With Two Events

The transition condition from state 1 to state 2 is: e1 COMPMULT e2. e1 and e2 are expressed as explained above. If the system is in state 1 (as in the example) and the composed condition is true the system will switch to state 2. If one of these two statements is not verified, this change of state is not carried out. The following algorithm illustrates the representation behavior of a composed condition using COMPMULT:

1. the operative part is computed;
2. the computational variables are assigned their new value;
3. the conditions of state change are consulted:
   (a) if the condition e1 COMPMULT e2 is verified then
      - the time increases to t + δt ;
      - the non-computational global variables are assigned their new value ;
      - the state change is carried out;
      - the system executes this principle in item - 1 - for the new state.

   (b) if the condition e1 COMPMULT e2 is not verified then
      - the time increases to t + δt ;
      - the non-computational global variables are assigned their new value ;

- the system stays in the same state ;
- the system revalues the condition with the same protocol as in - 3a - but without the revaluation of the non-computational variables.

6.2.2.2.3 PIPE Operator:

$$\text{CC} = SC_1 \text{ PIPE } SC_2.$$

CC is true when SC_1 is true and SC_2 becomes true before the life-time associated with the considered state has elapsed. We will see how to express this operation. We characterize CC according to different types SC_1 and SC_2.

- first case: $SC_1 = F(I_1, p_1)$ and $SC_2 = F(I_2, p_2)$ facts.

  We have particular relationships between I_1, I_2 and I_d (I_d was defined in section 5.1.2 and represents the time interval [t, t + ta(S)] where t is the input instant in the state S). Thus, with $I_1 = [t_1, t'_1]$, $I_2 = [t_2,t'_2]$ and I_d as defined before, in order to verify this association, we must have:

$$\begin{cases} I_1 \cup I_d \neq \emptyset \\ I_2 \cup I_d \neq \emptyset \quad ; \\ and\ t'_1 < t_2 \end{cases}$$

  to create the sequence of two facts;

- second case: $SC_1 = e(t_1, p_1)$ and $SC_2 = e(t_2, p_2)$ events.

  In order to verify this composition, we must have $t_1 < t_2$, in the same way t_1 and $t_2 \in I_d$ to create the sequence of two events;

- third case: $SC_1 = F(I_1, p_1)$ and $SC_2 = e(t_2, p_2)$ a fact and an event.

For this association, particular relationships exist between I_1, t_2 and I_d. Thus, with $I_1 = [t_1, t'_1]$ and I_d defined before, in order to verify this association, we must have:

$$\begin{cases} I_1 \cup I_d \neq \emptyset \\ t_2 \in I_d & ; \\ and\ t'_1 < t_2 \end{cases}$$

to create the sequence of a fact and an event ;

- fourth case: $SC_1 = e(t_1, p_1)$ and $SC_2 = F(I_2, p_2)$ an event and a fact. For this association, particular relationships exist between t_1, I_2 and I_d. Thus, with $I_2 = [t_2, t'_2]$ and I_d defined before, in order to verify this association, we must have:

$$\begin{cases} I_2 \cup I_d \neq \emptyset \\ t_1 \in I_d & ; \\ and\ t_1 < t_2 \end{cases}$$

to create the sequence of an event and a fact.

One illustration of the behavior of PIPE is shown below (the other possibilities can be found in Appendix A.1.3). For example, the condition for a transition is *cond = e1 PIPE e2* and drawn as in Fig. 6-17.

The transition condition from state 1 to state 2 is: e1 PIPE e2. e1 and e2 are expressed as explained previously. If the system is in state 1 (as in the example) and the composed condition is true the system will switch to state 2. If one of these two statements is not verified, this change of state does not occur. The following algorithm illustrates the representation behavior of a composed condition using PIPE:

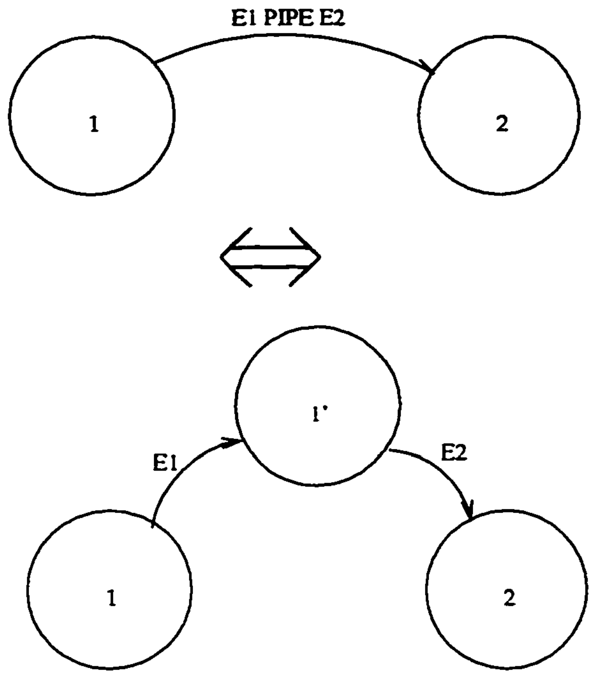


Figure 6-17: PIPE With Two Events

1. the operative part is computed;
2. the computational variables are assigned their new value;
3. the conditions of state change are consulted:
   (a) if the condition e1 is verified then
      - the time increases to t + δt ;
      - the non-computational global variables are assigned their new value ;
      - if the condition e2 is verified then
         - the time increases to t + δt ;
         - the state change is carried out;
      - if the condition e2 is not verified then
         - the time increases to t + δt ;
         - the system stays in the same state ;

(b) if the condition e1 is not verified then

- the time increases to t + δt ;
- the non-computational global variables are assigned their new value ;
- the system stays in the same state :
- the system revalues the condition with the same protocol as in item - 3a - but without the revaluation of the non-computational variables.

6.2.2.3 Priority Notion

All the reasoning about state change has been done for a single condition (simple or composed) implying a transition from one state to another. There is also a case in which several conditional arrows are associated with a state. Therefore, the possibility exists for going to several states. However, the pseudo-state graph is deterministic; it cannot have many possible transitions; so we must have a system with exclusive transitions. Thus, we introduce the priority notion onto each arrow. This notion signifies that if, at a given instant, we have several conditions verified for the considered state, we choose the arrow with the greatest priority. We represent this notion with a numeric value associated with each arrow (Fig. 6-18).

6.2.3 Translation in VHDL

This section presents the general structure of VHDL code generated from the internal model. The translation of basic elements defined in the internal model will be described. These basic elements are :

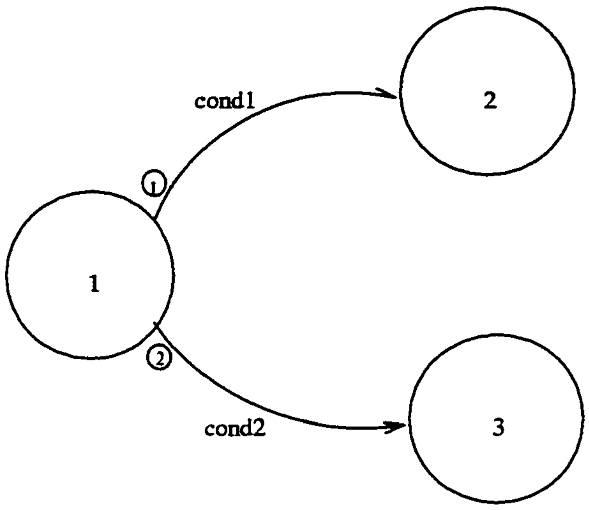


Figure 6-18: Priority Representation

- model states ;
- variables used in the specification language ;
- state changes ;
- global structure of VHDL code.

6.2.3.1 States in VHDL

In addition to the existing types in VHDL, a node type is defined corresponding to the node set S. To represent the set S in VHDL, for example S=$\{N_1, N_2, \ldots\}$, a type NODE is defined as:

$$\text{type NODE is } (N_1, N_2, \ldots).$$

A signal named STATE is of type NODE. With this signal, the system controls the change state and when a modification occurs in it, this causes the activation of the new node operative part. To change a new STATE value requires a specific process. This process is the main process and is in relationship with all processes implementing actions of each node. This relationship requires the declaration of the variable STATE as a signal because, in VHDL there is not other solution for exchanging information in the concurrent mode. Therefore, the signal STATE is declared as:

signal STATE : NODE { := initial value }.

In VHDL, the behavior of a transition from one node N to another, M, is as follows. The system is, at any given instant t, at a node N. In the main process, when the STATE assignment takes a new value M at t with STATE $\Leftarrow$ M, the system will be in this state at t + δt. This change implies a process activation containing the operative part of M. By convention, this process is called *state process* and its label is the name of the state itself. The structure of the state process is as follows:

```
State : process
        begin
        wait until (STATE = node);    – the process wait until it activation
        ⋮                             – action description of a node
        wait on STATE;
        end process;
```

6.2.3.2 Variable Description in VHDL

In the specification language, four types of variables are defined:

- input ;
- state ;
- computational ;

- output ;

In VHDL, by definition, input and output variables are included in the signal class. In addition, according to their particular utilization, state variables are also included in this class for the reason explained above. In VHDL, input and output signals are in the header of a VHDL code. However, state variables must be declared, and the syntax is :

signal signal_name : type { := initial value } ;

Computational variables are included in the variable class in VHDL. The behavior of this variable class is different than that of signals. Variables have no delay when there is an assignment. The syntax of this declaration type is as follows:

variable var_name : type { := initial value } ;

6.2.3.3 State Changes

The transition from one state to another is computed in a process named "main process". This process represents a VHDL description of internal and external functions from the internal model. In the main process, if the system is in a state characterized by a node, and a state change condition is verified, then the transition to another state (characterized by a new node or the same one) is computed. When the new node is known, the state process associated at the new node is activated. The main process perpetually scrutinizes the input and state variables. When conditions are verified, a node change (also known as system state change) occurs. It is this process which puts the system in a wait state when no transition conditions are verified. The main process functions as:

- the system is in a state characterized by a node N. Thus, the value of the STATE variable is N. The main process observes, at any given instant t, all variables used to create the transition conditions of the node N;

- if no conditions are true at this instant t, the main process (and thus the system) is in the **wait mode** realized by the instruction WAIT (i.e. structure of the main process);
- if one or more conditions are true at this instant t, the main process computes a state change assigning the signal STATE at a new state characterizing a node M led by a transition having the highest priority. Consequently, actions associated with M are evaluated. Thus, computational variables (variables in VHDL) and output variables (signals in VHDL) receive their new value respectively at t + δt and t+2δt.

The main process structure contains the instruction **CASE**. A CASE statement selects for execution one of a number of alternative sequences of statements: the chosen alternative is defined by the value of an expression. The expression must be of a discrete type, or of a one-dimensional character array type. The syntax of CASE is the following:

```
case_statement ::=
    case expression is
        case_statement_alternative
        { case_statement_alternative }
    end case ;
case_statement_alternative ::=
    when  choices =>
        sequence_of_statements
```

The main process structure is illustrated through a simple example. Let an internal model make up of three nodes N_1, N_2, N_3. Let transition conditions $cond_1, cond_2$ be associated respectively with a transition from N_1 to N_2 and from N_2 to N_3. The main process corresponding to this model is as follows:

```
main : PROCESS
BEGIN
  CASE STATE IS
       WHEN N_1 =>
             IF not(cond1)
                THEN
                      -- the system is in N_1
                      WAIT UNTIL (cond1);
                      -- If cond_1 is not true the system stays
                      -- in this state
             END IF ;
             IF (cond_1)
                THEN
                      -- If cond_1 is true
                      STATE <= N_2;
                      -- then the system changes state
                      -- represented by N_2
             END IF ;
       WHEN N_2 =>
             IF not(cond2)
                THEN
                      -- the system is in a state ofN_2
                      WAIT UNTIL (cond2)
                      -- If cond_2 is not true the system stays
                      -- in this state
             END IF ;
             IF (cond_2)
                -- If cond_2 is true
                THEN
                      STATE <= N_3;
                      -- then the system changes of state
                      -- represented byN_3
             END IF ;
  END CASE ;
END PROCESS main;
```

6.2.3.4 Global Software Structure

Having defined the basic elements of a VHDL description obtained from a design specification with the specification language, a global software structure can be proposed representing the place of these elements.

In the entity specification (ENTITY), all input and output variables are declared with their type and their input-output mode. The architecture specification (ARCHITECTURE) contains in it declarative part : variables, signals, constants used in all processes (main and state processes) and types defined as describing systems nodes. The body of the architecture contains the internal model view. All processes are included in a BLOCK structure. This BLOCK structure will be useful later when we want to insert the notion of hierarchy in the state graph representation of the specification language. Therefore, the global software will be :

```
ENTITY OF model-name IS
          PORT (variable - declaration[,variable - declaration] :
                 mode(I/O)
                 type;
                 [variable - declaration[,variable - declaration] :
                 mode(I/O)
                 type; ]
                 );
END model-name :
ARCHITECTURE behavioral OF model-name IS
declaration of signals, variables, constants, types
BEGIN
  BLOCK : {block - name}
  BEGIN
    main : PROCESS
    BEGIN
      CASE STATE IS
             WHEN N1 => ...
             WHEN N2 => ...
             :
      END CASE ;
    END PROCESS ;
    State1 : PROCESS
    BEGIN
      :
    END PROCESS state1;
    State2 : PROCESS
    BEGIN
      ⋮
    END PROCESS state2;
    ⋮
  END BLOCK ;
END behavioral;
```

6.2.4 Example of SPECIAL Front End

To illustrate SPECIAL and its translation process, the example of the RAM described in chapter 2 is used. Fig. 6-19 is a snapshot of the SPECIAL graphic interface.

As seen in chapter 5, two strategies are defined to perform this translation into a VHDL code. For the RAM, these two models are fully developed in the example in section 5.2.4 for the basic VHLLS model, and in the example in section 5.3.3 for the advanced VHLLS model.

The resulting VHDL code generated by SPECIAL is the following:

```
ENTITY OF RAM IS
          PORT (NRST : IN bit;
                   CS : IN bit;
                   RD : IN bit;
                   WR : IN bit;
                   AD : IN BIT_VECTOR(0to7);
                   DIN : IN BIT_VECTOR(0to3);
                   DOUT : OUT BIT_VECTOR(0to3);
                   READY : OUT bit
                   );
          CONSTANT T_READY_U : TIME := 60ns;
          CONSTANT T_READY_D : TIME := 1ns;
          CONSTANT T_ACCESS : TIME := 40ns;
          CONSTANT T_WRITE : TIME := 5ns;
END RAM ;
ARCHITECTURE behavioral OF RAM IS
CONSTANT nb_words : INTEGER := 2 * *8;
TYPE type_memory IS ARRAY(0 TO nb_words - 1) OF BIT_VECTOR(0 TO 3);
TYPE Type_stateis(Init0, Init1, Wait - st, R/W, R, W, Err);
SIGNAL state : Type_state := init0;
VARIABLE M : type_memory;
```

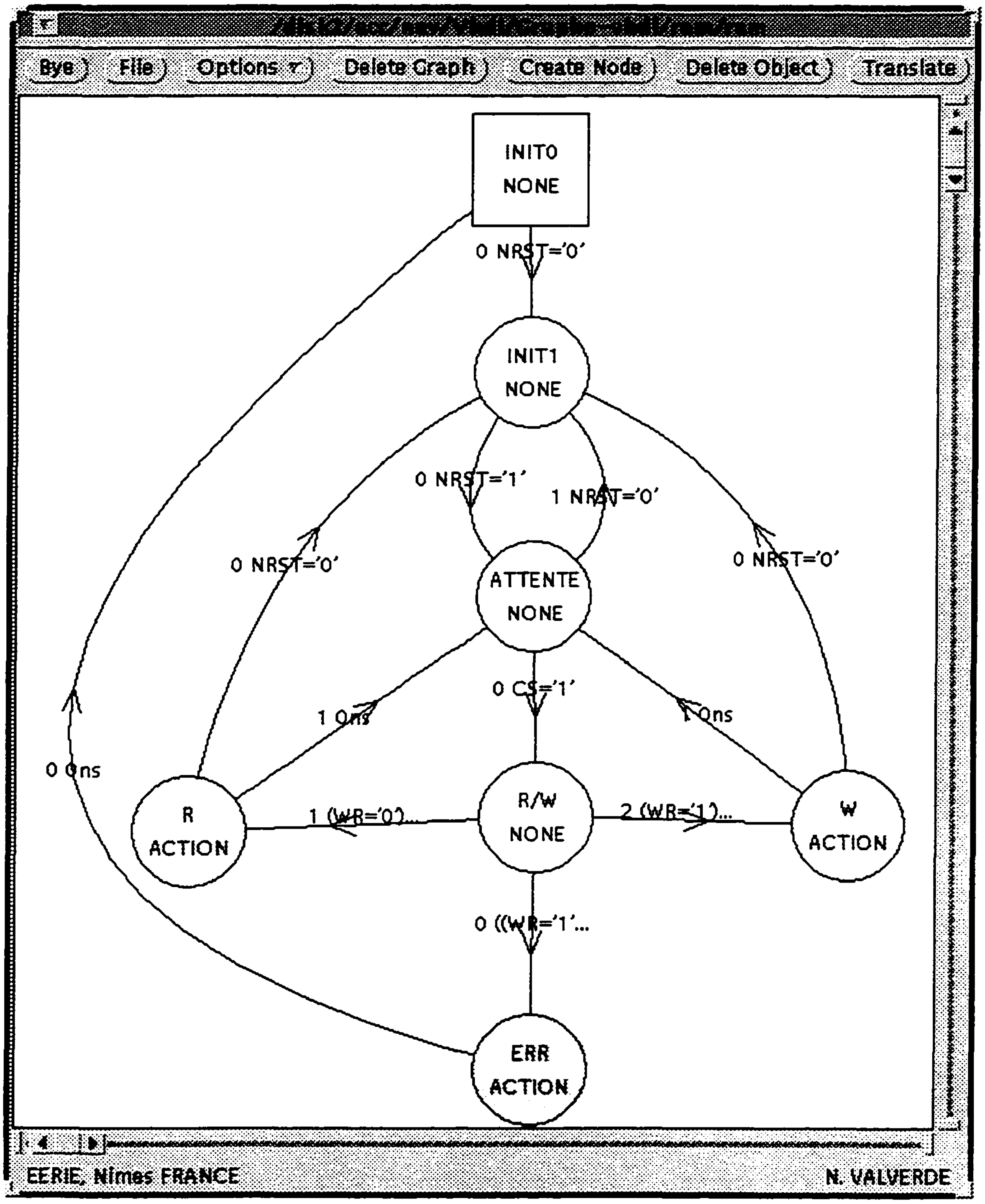


Figure 6-19: RAM Description With SPECIAL

```
FUNCTION value(bv : IN BIT_VECTOR(0 TO 7)) RETURN natural IS
VARIABLE n : NATURAL := 0;
BEGIN --process
   FOR l IN bv'low TO bv'high LOOP
         n := n * 2;
         IF bv(l) =' 1'
            THEN
                  n := n + 1;
         END IF ;;
  END LOOP;
   RETURN n;
END value;
BEGIN
  BLOCK : {RAM - Block}
  BEGIN
    main : PROCESS
    BEGIN
      CASE STATE IS
            WHEN Init0 =>
                     IF not(NRST =' 0')
                         THEN
                                  WAIT UNTIL NRST =' 0';
                     END IF ;
                     STATE <= Init1;
            WHEN Init1 =>
                     IF not(NRST =' 1')
                         THEN
                                  WAIT UNTIL NRST =' 1';
                     END IF ;
                     STATE <= Wait - st;
            WHEN Wait =>
                     IF not(CS =' 1')
                         THEN
                                  WAIT UNTIL CS =' 1';
                     END IF ;
                     STATE <= R/W;
            WHEN R/W =>
                     IF not((WR =' 0' AND RD =' 1') OR
                         (WR =' 0' AND RD =' 0') OR
                         (WR =' 1' AND RD =' 1') OR
                         (WR =' 1' AND RD =' 0'))
                         THEN
                                  WAIT UNTIL ((WR =' 0' AND RD =' 1') OR
                                  (WR =' 0' AND RD =' 0') OR
                                  (WR =' 1' AND RD =' 1') OR
                                  (WR =' 1' AND RD =' 0'));
                     END IF ;
                     IF (WR =' 0' AND RD =' 1')
```

```
                    THEN
                                $state \Leftarrow R$;
                ELSIF $(WR =' 1'$ AND $RD =' 0')$
                        THEN
                                $state \Leftarrow W$;
                ELSIF $(WR =' 1'$ AND $RD =' 1')$
                        THEN
                                $state \Leftarrow Err$;
                ELSIF $(WR =' 0'$ AND $RD =' 0')$
                        THEN
                                $state \Leftarrow Err$;
                END IF ;
        WHEN $R \Rightarrow$
                IF $not(NRST =' 0')$
                    THEN
                                WAIT UNTIL $NRST =' 0'$ FOR $1ns$;
                END IF ;
                IF $NRST =' 0'$
                    THEN
                                $STATE \Leftarrow Init1$;
                      ELSE
                                $STATE \Leftarrow WAIT$;
                END IF ;
        WHEN $W \Rightarrow$
                IF $not(NRST =' 0')$
                    THEN
                                WAIT UNTIL $NRST =' 0'$ FOR $1ns$;
                END IF ;
                IF $NRST =' 0'$
                    THEN
                                $STATE \Leftarrow Init1$:
                      ELSE
                                $STATE \Leftarrow WAIT$;
                END IF ;
        WHEN $Err \Rightarrow$
                IF $not(true)$
                    THEN
                                WAIT  FOR $1ns$;
                END IF ;
                $STATE \Leftarrow Init0$;
    END CASE ;
END PROCESS ;
$R - p$ : PROCESS
BEGIN
    WAIT UNTIL $(STATE = R)$;
    $Ready \Leftarrow' 1'$ AFTER $T - READY - U$,
    $'0'$ AFTER $T - READY - U + T - READY - D$;
    $DOUT \Leftarrow M(value(AD))$ AFTER $T - ACCESS$;
    WAIT ON STATE;
  END PROCESS R - p;
  W - p : PROCESS
  BEGIN
    WAIT UNTIL (STATE = W);
    M(value(AD)) <= DIN AFTER T - WRITE;
    WAIT ON STATE;
  END PROCESS W - p;
  Err - p : PROCESS
  BEGIN
    WAIT UNTIL (STATE = Err);
    ASSERT FALSE
    REPORT "Wrong values for WR and RD when CS rises"
    SEVERITY WARNING;
    WAIT ON STATE;
  END PROCESS W - p;
 END BLOCK ;
END behavioral;
```

6.3 Conclusions on SPECIAL

In Section 1.1.2, for meeting the challenges about defining a new generation of CAD tools, the following enumeration was proposed:

1. the hypothetical introduction of a new design process using a generalized synthesis approach as shown in Fig. 1-4. The emphasis in this thesis is on the front-end synthesis, called VHLLS;

2. if (1) is proven then the next generation of design automation tools is introduced as a practical consequence of a generalized synthesis process;

3. therefore, the complexity of microelectronics systems design is lessened, or at least maintained, by starting a design process at a higher level of abstraction;

4. and, a high-level specification is incorporated as the entry level in an automated design flow.

Item (1) represents our primary objective. Item (2) introduces the issue of feasibility to this problem. Therefore, the hypothesis of the research problem can be stated as follows: having (1), we can define (2) or mathematically ((1) $\Rightarrow$ (2)). So, Chapters 2 and 3 identify the need of introducing a new generation of CAD tools and define a formalism to characterize them. Chapter 4 reviews exciting methods in order to identify description methodologies which can be classified as the next generation of CAD tools. To have ((1) $\Rightarrow$ (2)) true, Chapters 5 and 6 define a framework enabling system specifications in a graphical manner and a translation process of these specifications allowing the generation of the system at a lower level of abstraction. As a result, the implication of getting (3) and (4) has been partially demonstrated. Indeed, to keep this problem feasible, the domain of investigation was restricted to a minimal configuration of the design space referred to as C_{min} (it is the minimal set of characteristics, a next generation CAD tool must meet).

$C_{min} = \{$

Characteristics	Checkmark
Sequentially Decomposable Activities	√
Concurrently Decomposable Activities	
State Transitions	√
Immediate Mode Change	
Activity Completion	√
Delay Specification	√
Asynchronous Activities	
Design for { Testability, Manufacturing, etc }	
Multiple Model Representations	
Reusability	

$\}$

Table 6.1: First Generation VHLLS Characteristics

We can state then:

Theorem 6.1 *Under the minimal configuration defined by C_{min}, the following relation is* *verified by the SPECIAL environment:*

$$(hypothesis((1) \Rightarrow (2))) \Rightarrow ((3) \wedge (4))$$

Proof: This thesis constitutes the proof.

As an illustration for this new generation of CAD tools, next chapter is advocated to illustrate SPECIAL using three examples.

Chapter 7

VHLLS Examples

The previous chapters introduce the notions needed to implement VHLLS. As a result, a CAD tool called Specification Procedure for Electronic Circuits in Automation Language (SPECIAL) is defined and implemented as illustrated in Chapters 5 and 6. This chapter illustrates SPECIAL using three examples. The first one shows the method of capturing specifications using SPECIAL. The second example identifies where SPECIAL fits in a real design flow. The last example illustrates some limitations of this first version of SPECIAL.

7.1 Process Controller

The behavior of a computer system can be described as a set of asynchronous, concurrent, and interactive processes, where a process for this example is viewed as a device defined as an identifiable sequence of related actions. This process performs a single execution of a program. It can be in one of these major states:

- Busy or executing;
- Idle and ready to begin execution;

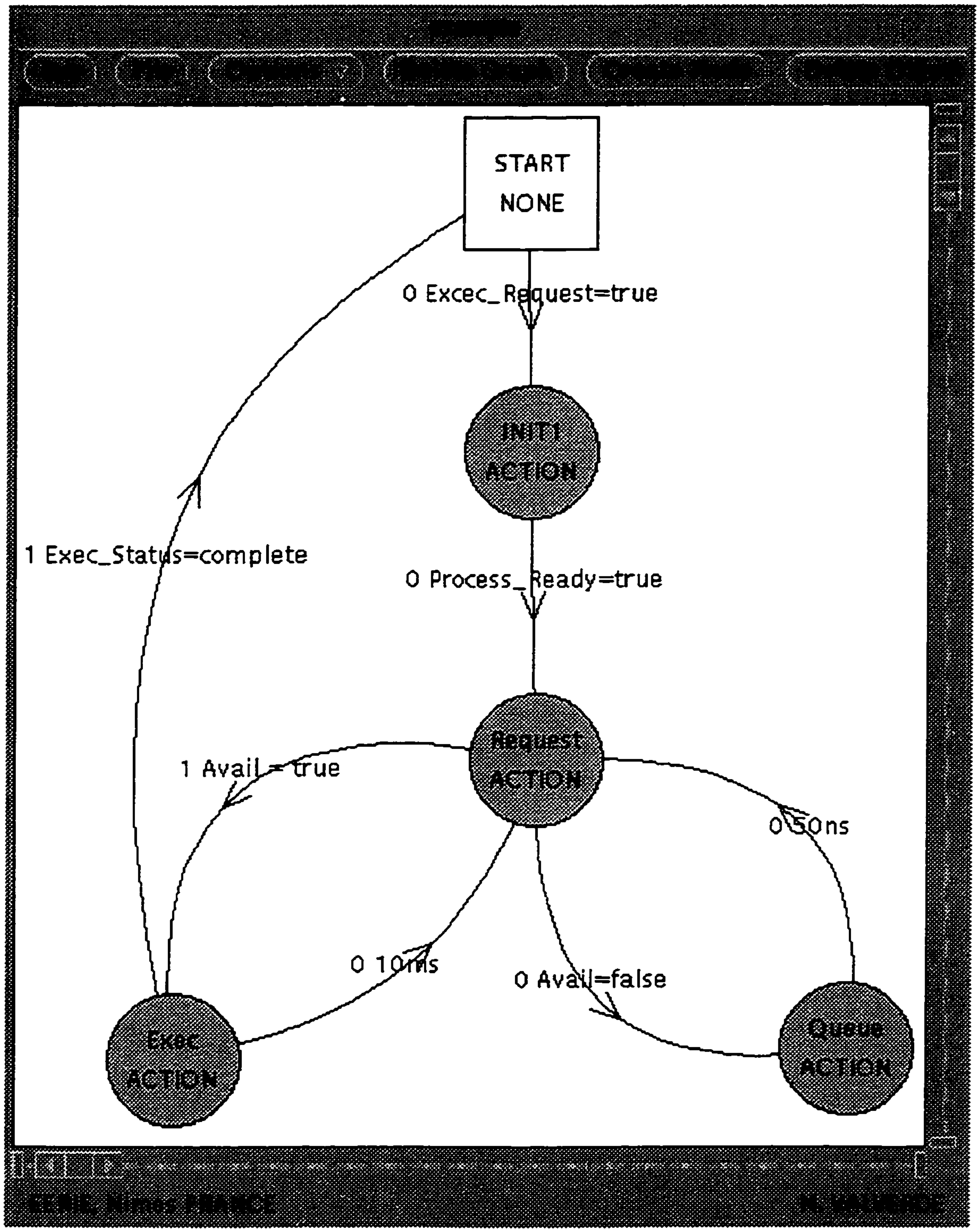


Figure 7-1: Process Controller Specification

- Idle while execution is temporarily suspended;

- Idle but not ready to begin execution.

This process uses shared system resources. The execution of a process is suspended if a resource it requires has been preempted by other processes. To make sure that a process is properly executed, a process controller must verify that all the resources are available before telling the process to start the execution of the program. The time allocated for the program execution must not exceed 10ms. If the resources are not available, the process makes a request for them and samples their availability every 50ns until all the resources are ready to deliver their services. One approach to specifying the behavior of this process controller is shown in Fig. 7-1.

The translation process led to the following VHDL description:

```
PACKAGE process_ctrl_package IS
            CONSTANT nb_process : integer := 4;
            TYPE status IS (not_available, available);
            TYPE Resource_status_type IS array(0 TO nb_process) OF status;
            TYPE Process_ID IS
            RECORD
                        ID : bit_vector(3 DOWNTO 0);
                        intpt : bit;
            END ;
            TYPE Request_resources_type IS array(0 TO nb_process) OF Process_ID;
            TYPE Active_process_type IS (Idle, Fast_init, Full_init, Start, Resume, Stop);
            TYPE Exec_Status_type IS (Idle_Init, Idle_suspended, Busy, Completed,
            Error, Power_on);
END process_ctrl_package;
USE work.process_ctrl_package.all;
ENTITY ctrl_process IS
          PORT
              (
              Exec_Request, Process_ready : IN BOOLEAN;
              Acknowledge_request : OUT BIT;
              Resources_status : IN Resource_status_type;
              Exec_status : IN Exec_Status_type;
              PROCESS_ID : IN bit_vector(3 DOWNTO 0);
              Request_resource : OUT Request_resources_type;
              PROCESS_Init : OUT Active_process_type;
              Active_process : OUT Active_process_type
```

```
                );
END ;
ARCHITECTURE COMP OF ctrl_process IS
BEGIN
  GRAPHE : BLOCK
  TYPE NOEUD IS
  (
  START,
  INIT1, Exec, Queue, Request
  );
  SIGNAL ETAT : NOEUD;
  SIGNAL
  Avail : BOOLEAN;
  BEGIN
    PRINCIPAL : PROCESS
    BEGIN
      CASE ETAT IS
            WHEN INIT1 =>
                    IF not(PROCESS_Ready = true)
                        THEN
                                WAIT UNTIL (Process_Ready = true);
                    END IF ;
                    ETAT <= Request;
            WHEN Exec =>
                    IF not(Exec_Status = Completed)
                        THEN
                                WAIT UNTIL (Exec_Status = Completed) FOR 100ns;
                    END IF ;
                    IF (Exec_Status = Completed)
                        THEN
                                ETAT <= START;
                         ELSE
                                ETAT <= Request;
                    END IF ;
            WHEN Queue =>
                    IF not(true)
                        THEN
                                WAIT UNTIL false FOR 50ns;
                    END IF ;
                    IF (false)
                        THEN
                                null;
                         ELSE
                                ETAT <= Request;
                    END IF ;
            WHEN Request =>
                    IF (not(Avail = true))
                        THEN
                                WAIT UNTIL Avail = true FOR 0ns;
                    END IF ;
                    IF ((Avail = true))
                        THEN
                                ETAT <= Exec;
                         ELSE
```

```
                                ETAT <= Queue;
                         END IF ;
                    WHEN START =>
                         IF (not(Exec_Request = true))
                            THEN
                                 WAIT UNTIL (Exec_Request = true);
                         END IF ;
                         ETAT <= INIT1;
   END CASE ;
 END PROCESS PRINCIPAL;
 Exec_st : PROCESS
 BEGIN --Exec_st
   WAIT UNTIL (ETAT = Exec);
   IF (Exec_status = Idle_init)
       THEN
              Active_process <= Start;
   ELSIF (Exec_status = Idle_suspended)
          THEN
                 Active_process <= Resume;
           ELSE
                 ASSERT false
                 REPORT "Error : Process found BUSY when it should be IDLE"
                 SEVERITY ERROR;
                 Active_process <= full_init;
   END IF ;
   WAIT ON ETAT;
 END PROCESS Exec_st;
 INIT1_st : PROCESS
 BEGIN --INIT1_st
   WAIT UNTIL (ETAT = INIT1);
   IF ((Exec_status = Idle_suspended) OR (Exec_status = Idle_init))
       THEN
              Process_Init <= fast_init;
   ELSIF ((Exec_status = Power_on) OR (Exec_status = Error))
          THEN
                 Process_Init <= full_init;
           ELSE
                 Process_Init <= idle;
   END IF ;
   Acknowledge_request <='1','0' AFTER 10ns;
   WAIT ON ETAT;
 END PROCESS INIT1_st;
 Queue_st : PROCESS
 BEGIN --Queue_st
   WAIT UNTIL (ETAT = Queue);
    FOR i IN 0 TO nb_process LOOP
          IF (Resources_status(i) = not_available)
              THEN
                     Request_resource(i).ID <= process_ID;
                     Request_resource(i).intpt <='1','0'after20ns;
          END IF ;
   END LOOP;
   WAIT ON ETAT;
 END PROCESS Queue_st;
```

```
      Request_st : PROCESS
      VARIABLE i : integer;
      BEGIN --Request_st
        i := 0;
        WAIT UNTIL (ETAT = Request);
        while (i <= nb_process) LOOP
              IF (Resources_status(i) = not_available)
                 THEN
                        Avail <= false;
                  ELSE
                        i := i + 1;
                        Avail <= true;
              END IF ;
        END LOOP;
        WAIT ON ETAT;
      END PROCESS Request_st;
    END BLOCK GRAPHE;
END COMP;
```

Notice that some sections of this above VHDL code is not yet automated. For example, the first part of the code referred to as "package" must be defined by the designer because this section of code allows the designer to define the type of each input or output.

7.2 SPECIAL in MCM Design Flow

An example is selected from a real project involving satellite development. Three industrial design methods and SPECIAL are then used to implement that same example to obtain a framework for comparison.

The UNH's Institute for the Study of Earth, Ocean and Space (EOS) is building a light satellite [FOR94] to understand the origin of Gamma Ray Bursts (GRBs). The Cooperative Astrophysics and Technology SATellite (CATSAT) is a small space flight mission designed to better understand this phenomenon using a multi-observation approach. The general configuration of CATSAT includes:

- a set of sensors able to detect GRBs and to capture relevant parameters;
- a communication device.

The CATSAT scientific instruments sort and store information from the sensors into appropriate memory locations with three major subsystems: an Analog Electronics Unit (AEU); a Digital Electronics Unit (DEU); and an Automatic Gain Control system (AGC). The AEU prepares analog signals from individual sensors for conversion to digital channel signals. The DEU accepts converted digital pulse amplitudes and sorts them by channel into corresponding spectra. The AGC consists of gain control elements; each element regulates the gain of a specific sensor. The AGC performs continuous sensor calibration to ensure accurate measurements over time. This is accomplished by comparing sensor gains to the reference energies of radioactive source photons. Each sensor gain is controlled by a gain control element. This element consists of an up-down counter and a digital-to-analog converter. The counter stores the value which is directly proportional to the gain. The value drives a digital-to-analog converter regulating the sensor gains.

The Up-Down Counter which was selected for the presented experiment, has the following specification:

> The up-down counter is a synchronous digital circuit which increments or decrements its output every T_Trig period of time. A reset command can be applied any time to initialize the counter.

The presented up-down counter, depicted in Fig.7-2, is suitable for SPECIAL. Four different approaches are investigated. The respective entry levels are:

1. Specification level using SPECIAL to synthesize into a behavioral description;
2. RTL behavioral level;
3. Gate level;
4. Layout level (VLSI).

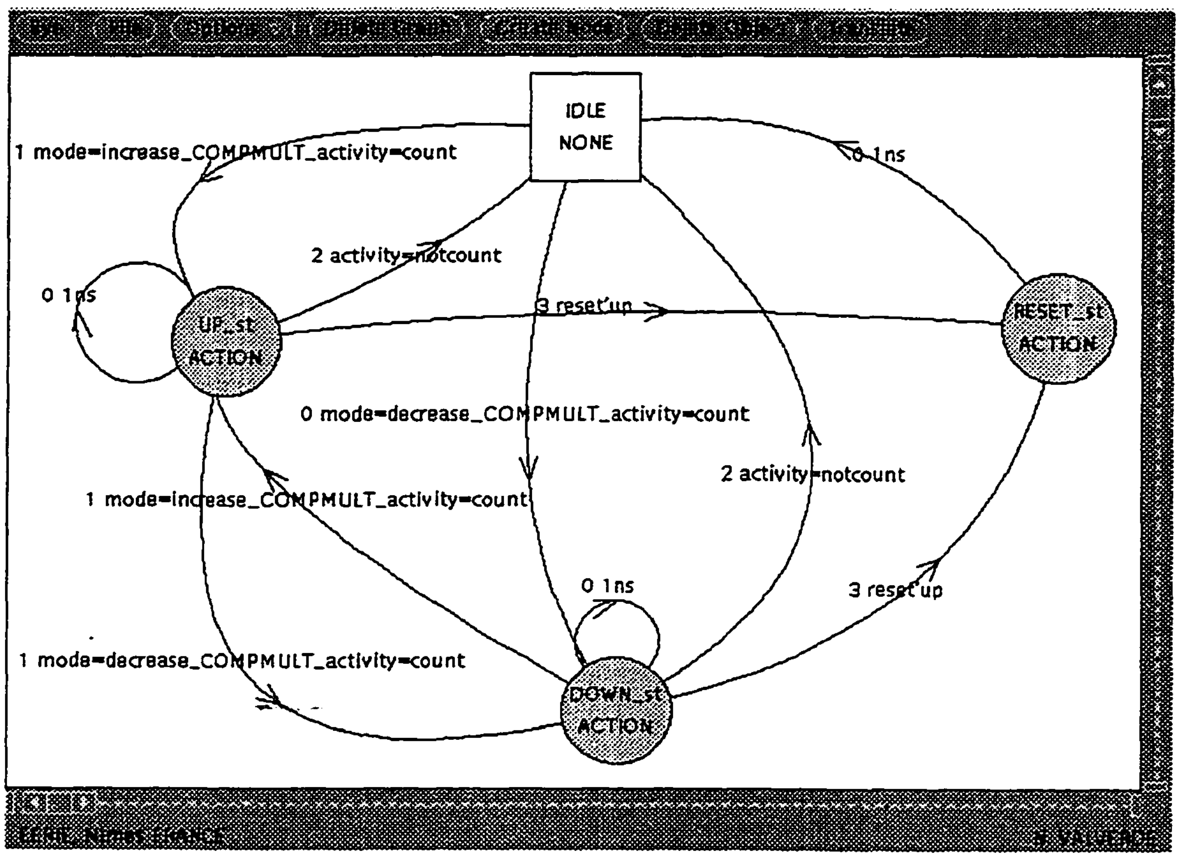

IDLE
NONE
1 mode=increase_COMPMULT_activity=count
0 1ns
2 activity=notcount
0 1ns
UP_st
ACTION
3 reset'up
RESET_st
ACTION
0 mode=decrease_COMPMULT_activity=count
2 activity=notcount
1 mode=increase_COMPMULT_activity=count
0 1ns
3 reset'up
1 mode=decrease_COMPMULT_activity=count
DOWN_st
ACTION

Figure 7-2: Up-Down Counter For CATSAT

In all of these methods as illustrated in Fig. 1-4, the targeted technology is MCM and the use of synthesis processes is prioritized [HRVJ95]. To generate the MCM layout, two methods are used: manual and automatic. The design environment used is provided by Mentor Graphics and the list of used tools includes: Design Architect, Quick VHDL, QuickSim II, Autologic, IC Station, MCM station [Cor95].

The selected example has been implemented using four design flows. Results and analysis are presented in order to position SPECIAL among the three others methods. The use of synthesis processes in the Mentor Graphics^{TM} design environment has been practiced as much as possible. The highest level which can be synthesized is RTL with VHDL assistance. Therefore, the VHDL code generated by SPECIAL needs to be adjusted to the requirement

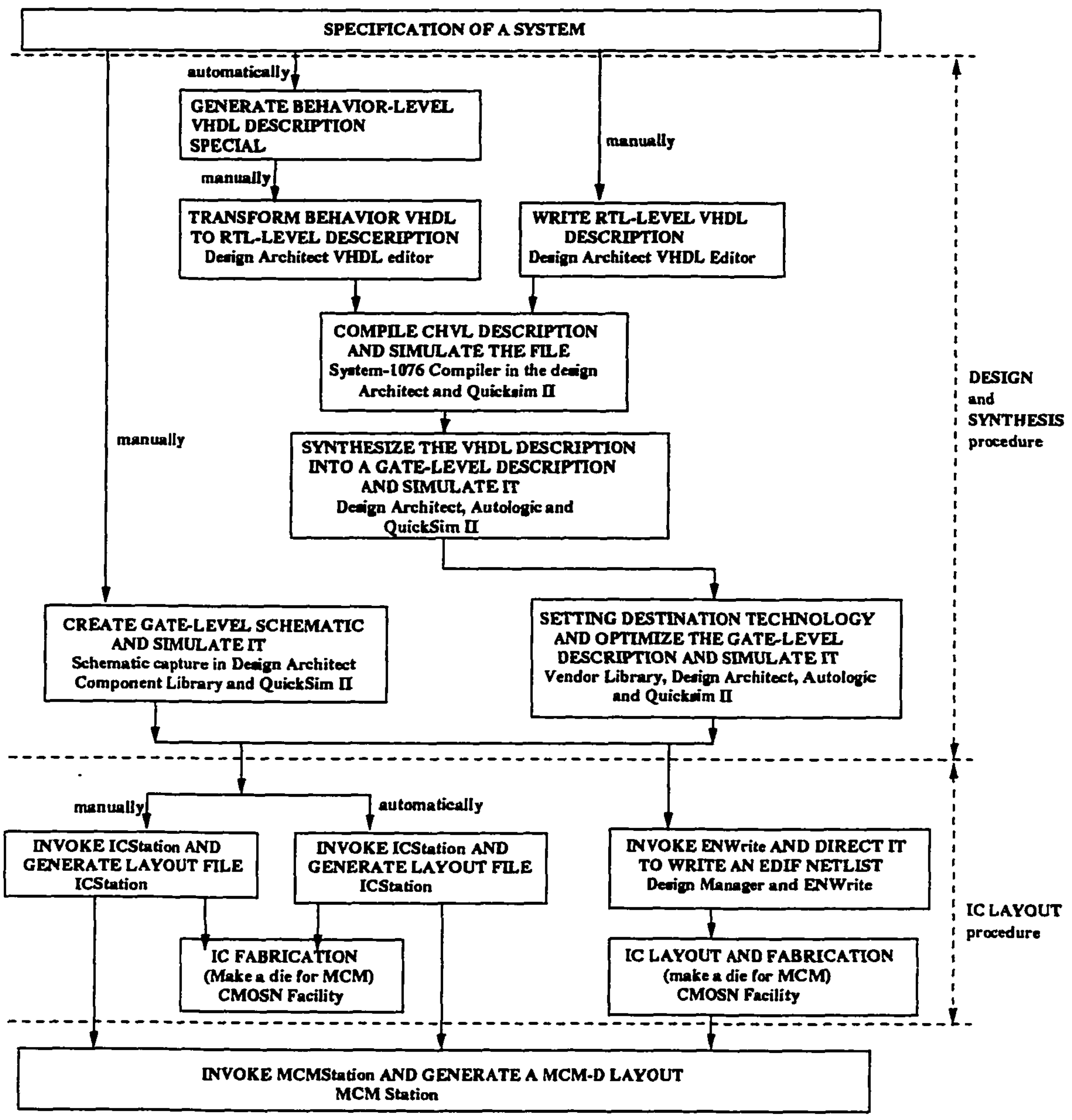

SPECIFICATION OF A SYSTEM
automatically
GENERATE BEHAVIOR-LEVEL VHDL DESCRIPTION SPECIAL
manually
TRANSFORM BEHAVIOR VHDL TO RTL-LEVEL DESCERIPTION Design Architect VHDL editor
manually
WRITE RTL-LEVEL VHDL DESCRIPTION Design Architect VHDL Editor
COMPILE CHVL DESCRIPTION AND SIMULATE THE FILE System-1076 Compiler in the design Architect and Quicksim II
manually
SYNTHESIZE THE VHDL DESCRIPTION INTO A GATE-LEVEL DESCRIPTION AND SIMULATE IT Design Architect, Autologic and QuickSim II
CREATE GATE-LEVEL SCHEMATIC AND SIMULATE IT Schematic capture in Design Architect Component Library and QuickSim II
SETTING DESTINATION TECHNOLOGY AND OPTIMIZE THE GATE-LEVEL DESCRIPTION AND SIMULATE IT Vendor Library, Design Architect, Autologic and Quicksim II
DESIGN and SYNTHESIS procedure
manually
automatically
INVOKE ICStation AND GENERATE LAYOUT FILE ICStation
INVOKE ICStation AND GENERATE LAYOUT FILE ICStation
INVOKE ENWrite AND DIRECT IT TO WRITE AN EDIF NETLIST Design Manager and ENWrite
IC FABRICATION (Make a die for MCM) CMOSN Facility
IC LAYOUT AND FABRICATION (make a die for MCM) CMOSN Facility
IC LAYOUT procedure
INVOKE MCMStation AND GENERATE A MCM-D LAYOUT MCM Station

Figure 7-3: Up-Down Counter: Design Methodologies

of RTL. Basically, the structure of the resulting code is the same. A **clock** signal has to be explicitly defined and all processors have to be guarded with control signals such as *reset, clock, activity mode,* and *state variable.* An additional process has to be implemented to manage the next state transition. Having given that new description, a synthesis process is applied to generate a gate level description with some additional constraints such as the width of counter output (12 bits). Using the Mentor Graphics' IC Station, the layout is automatically generated using the standard CMOSN library. Autoplace and autoroute rules are defined within the library and sufficiently generate the layout. The layout is invoked in the MCM environment and creates a *die.* The last step is to implement the counter as an MCM board allowing a single package with multiple dies (10) (limited by the number of I/O pins of the MCM package (172)). Using the same principles and the same set of tools, three other designs were completed for the up-down counter:

- specification capture and a synthesis process to obtain the MCM implementation;
- an RTL-level description and a synthesis process;
- a schematic description to exercise optimization followed by synthesis;
- an IC layout description generated from a non-optimized schematic.

For a simple design such as the up-down counter, the accurate time spent is difficult to determine since there was a learning curve to become familiar with the tools. Therefore, all time comparisons provided below are relative. Let us label:

- *method1* as the MCM circuit generated through SPECIAL;
- *method2* as the MCM circuit obtained from the RTL description;
- *method3* as the MCM circuit resulting from the schematic description;

- *method4* as the MCM circuit drawn from the IC layout description.

	VHDL lines	Gates	Transistors
Method1		196	1258
Method2	27	161	1102
Method3	-	61	622
Method4	-	70	708

Table 7.1: Design Sizes

	VHDL Behavior	VHDL RTL	Gates	Transistors	MCM	Approx. total time spent
Method1	1 + analysis	4	1	1	20	27 + analysis
Method2	-	3 + analysis	1	1	20	25 + analysis
Method3	-	-	80 + analysis	1	20	101 + analysis
Method4	-	-	-	160 + analysis included	20	180

Table 7.2: Design Timing in Hours

Table 7.1 shows the size of the designs which clearly depends on the level of abstraction. One can notice that method3 has the smallest number of transistors. The reason is that the optimization at that level is well understood. Also, during the design process at the gate level, designers used some ad-hoc optimization features complemented with those from the Mentor Graphics^{TM} synthesis tool. Moreover, the time spent (see Table 7.2) is substantial in comparison with the VHDL behavioral level method.

For the SPECIAL-oriented method1, the VHDL generation at the behavioral level is very efficient. However, because no synthesis tool commercially exists to translate from that level to the RTL level, this transformation has to be accomplished manually increasing the design time. Methodl and method2 generate the design in a relatively short time but with roughly twice the components. For prototyping, however, these two methods are sufficient. The last method can be considered the worst because it consumes substantially more time to sketch manually the design layout and has a larger number of transistors.

The described activities involved five students [JIA95, HEI95]. The evaluation of the above approaches shows that the optimal method to design a circuit depends upon the purpose of the circuit (prototype, final design). The SPECIAL tool is efficient to capture specifications and translate them into a high-level behavioral description. However, there is a need for a synthesis process to generate an RTL level code. With this gap filled, a top-down design process can be performed automatically from the specification directly to an MCM design. That would be very suitable for optimizing the prototype synthesis process.

7.3 Sample of Non Working Features in SPECIAL

The complexity of specification is limited with the first version of SPECIAL. Hierarchy enabling multiple design descriptions is not defined as well as concurrently decomposable activities. Indeed, for each state, the only representation to describe sub-activities is the sequential syntax of VHDL. For example, if a designer wants to describe a simple computer system, he (or she) is not able to specify it in a same tool environment session. No features allows the designer to say that a computer system is composed by a CPU , a clock generator and an input/output interface. These three sub-systems are indeed behaving concurrently. Also, if the designer wants to specify the behavior of the CPU, a first restriction is that only the sequential syntax of VHDL can be used and so it is not possible to describe a sub-unit of the CPU such as the memory unit and the execution unit which are concurrent sub-systems. Such a description can be easily described using a formalism such as SpecCharts [VNG91b]

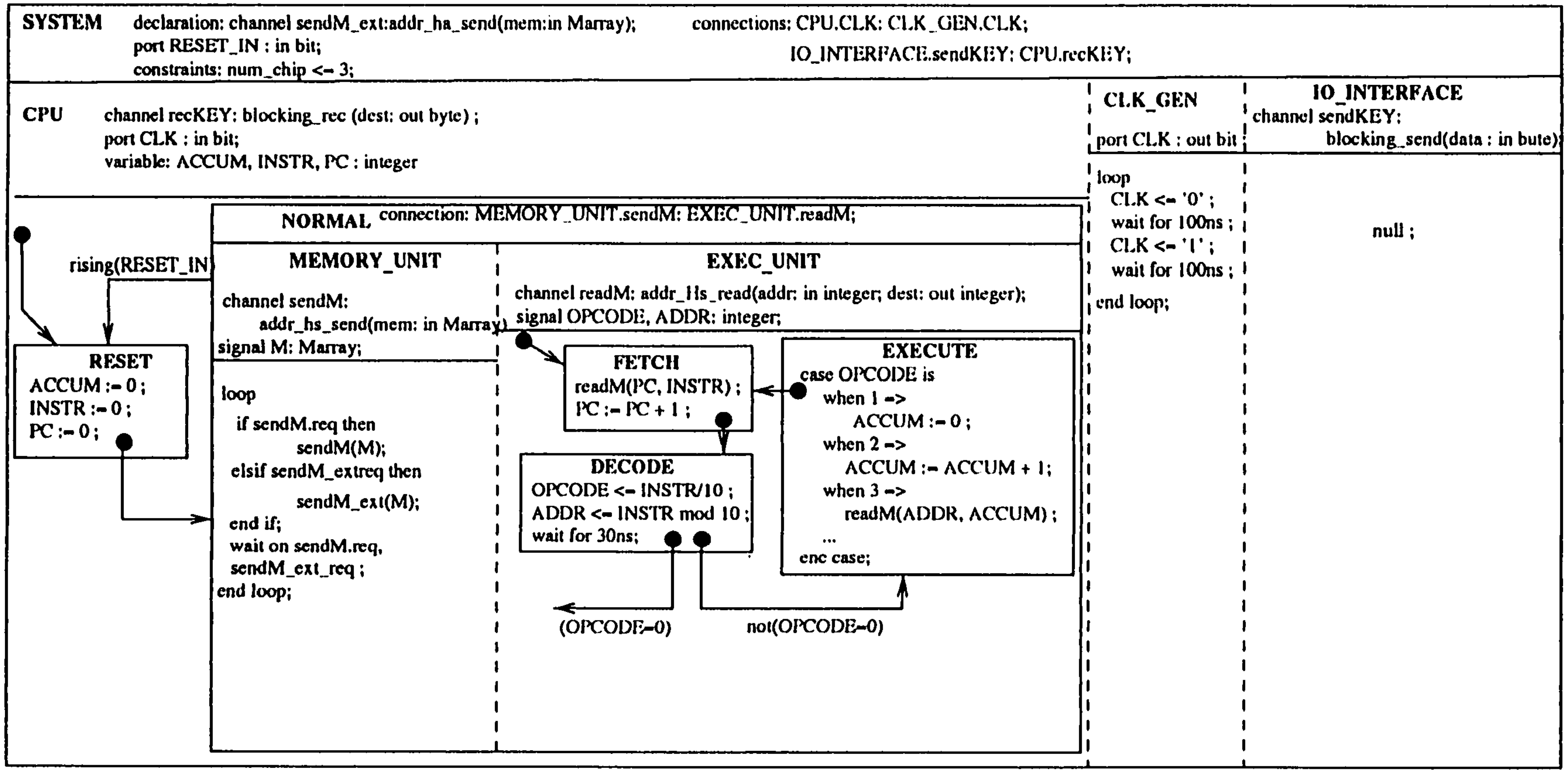


Figure 7-4: Graphical SpecChart of a Simple Computer System

as show in the Fig. 7-4.

As demonstrated in chapter 4, SpecChart is the most advanced tool to capture system specification. However, SpecChart is not able to describe directly specification given in section 7.1. The designer would have to go through a refinement stage in order to meet the syntax of SpecChart.

7.4 Closing Remarks

The first version of SPECIAL is a prototype which applies the simplest form of the VHLLS process. The chapter has illustrated its strengths as well as its weaknesses. Further developments are needed to comply to the ultimate goal of having a VHLLS able to generate a behavioral description from specifications described in many ways as defined in Chapter 1.

Chapter 8

Conclusions and Future Plans

8.1 Conclusions

This thesis is organized into two parts. The first part is the most important because it focuses on capturing the evolution of the design process in the design space. To perform this task, a good understanding of the design space is required in order to characterize this evolution. As a result, a formalism is proposed to model the design space and any transformation processes in this space. For example, synthesis processes can be represented by a mathematical notation following formal rules. Also, this formal model of the design space allows the definition of metrics such as the distance of an evolution, the cardinal of a tool characteristics set, etc... Using the above formalism, the current status of CAD tools can be characterized and the use of the metrics allows an immediate comparison. Furthermore, the same formalism allows the identification of the next generation of CAD tools by highlighting characteristics which are not met by available tools. Having specified the next generation of CAD tools, an evolution mechanism in the design space is defined and referred to as VHLLS. Therefore, the entry description method of these tools can be automatically transformed in a description accepted by today's CAD tools.

From the set of characteristics defining the next generation of CAD tools, it was noted in Chapter 4 that none of the description methods meet a characteristic called "Delay specification". This characteristic states that a designer can specify a time constraint allowing the system to change its state automatically after a certain duration. Once this characteristic has been identified, a minimal configuration is chosen to include the "Delay specification" characteristic. This minimal configuration is sufficient to demonstrate the feasibility of encapsulating time in the description model. So, a CAD tool called SPECIAL has been realized to implement the characteristics specified by this minimal configuration for the next generation of CAD tools.

8.2 Future Developments

Future developments need to focus on the next generation of VHLLS. This statement implies that the comparison metrics for the next generation of CAD tools need to be refined either by adding more characteristics, by ordering characteristics using a weighting scheme (to be defined) or by developing a hierarchy of characteristics. Any of these refinements on the characteristics set for the next generation of CAD tools will have direct impact on the evolution of VHLLS. For example, refining the characteristic "design for test" will lead to a VHLLS which could generate Boundary Scan Description Language (BSDL) code. With the same characteristic, another problem in testing micro-electronics devices is to generate test patterns from a behavioral description of a system. So, the next generation of CAD tool could help capturing the specification of that system and automatically generate behavioral test patterns instead of extracting them from its behavioral description [SCG93]. Another characteristic to address is the physical reconfigurability of computers. It is not clear yet if the characteristics set introduced in this thesis contains all the elements to characterize this future design aspect.

Considering the formal description of the design space as defined in this thesis, another future. development will be to implement the second generation of SPECIAL. The first step will be to implement the knowledge base and its knowledge manipulation mechanisms. Thereafter, the other crucial characteristic to encapsulate in SPECIAL will be the "multi-model representation" because existing description methods will have a framework to be integrated in SPECIAL, increasing then the number of characteristics met.

The implementation of the previous suggestions will define the second version of SPECIAL. This version will enable design cycle experiments. The conclusions of these experiments will draw a road map for further developments.

Bibliography

[AB94] Péter ARATO and István BERES. A high-level datapath synthesis method for pipelined structures. *Microelectronics Journal*, 25(3), 1994.

[AF94] J. ALLEN and G. FERGUSON. Actions and events in interval logic. Technical report, The University of Rochester, NY, 1994.

[ALL84] J. ALLEN. Towards a general theory of actions and time. *Artificial Intelligence*, 23:124–154, july 1984.

[BCM+88] R. BRAYTON, R. CAMPOSANO, G. De MICHELI. R. OTTEN, and J. Van EIJNDHOVEN. *The Yorktown Silicon Compiler System*. Silicon Compilation. Addison-Wesley, 1988.

[BLA97] D. BLANCHARD. PLDs and FPGAs: A market report. *Printed Circuit Design*, 14(8), August 1997.

[BS91] D. BELINA and A. SARMA. *SDL with Application from Protocol Specifications*. Prentice Hall, 1991.

[CLA73] C. CLARE. *Designing Logic Systems using State Machine*. McGraw-Hill Inc., 1973.

[Cor93a] Programmable Electronics Performance Corp. Benchmark suite #1, version 1.2, March 1993.

[Cor93b] Mentor Graphics Corporation. MCM station. Software Version 8.2_5, 1993.

[Cor95] Mentor Graphics Corp. Homepage: http: //www. mentorg. com/. Software Version 8.2_5, 1995.

[CYR94] W. CYRE. Conceptual representation of waveform for temporal reasoning. *IEEE transaction on computers*, 43(2):186–200, Febrary 1994.

[DGLW92] N. DUTT, D. GAJSKI, S. LINAND, and A. WU. *High-Level Synthesis : Introduction to Chip and System Design.* Kluwer Academic Publishers, Boston, Massachusetts, 1992.

[DON96] M. DONLIN. Graphical-code generators ease path the HDL design. *Computer Design*, pages 94–99, Dec. 1996.

[FOR94] David J. FORREST. *CATSAT Proposal.* UNH, 1994.

[FRE85] M. FRENCH. *Conceptual Design for Engineers.* Design Council Books, London, UK, 2nd edition, 1985.

[GHR93] D. GABBAY, I. HODKINSON, and M. REYNOLDS. *Temporal Logic: Mathematical Foundations and Computational Aspects.* Oxford University Press, 1993.

[GK83] D. GAJSKI and R. KUHN. Gest editor's introduction: New VLSI tools. *IEEE Computer*, 16(12):11–14, Dec. 1983.

[GOL70] A. GOLDMAN. *A Theory of Human Action.* Prentice-Hall, Englewood Cliffs, NJ, 1970.

[GVN93] D. GAJSKI, F. VAHID, and S. NARAYAN. SpecCharts: a VHDL front-end for embedded systems. Technical Report 93-31, University of California, Irvine, June 1993.

[HCG93] T. HADLEY, V. CHAIYAKUL, and D. GAJSKI. A data structure for interactive synthesis. Technical Report 93-6, Info. and Computer Science Dept., UCI, January 1993.

[HEI95] K. HEIN. A theoretical and practical approach to multichip module design. Master's thesis, University of New Hampshire, December 1995.

[HIL85] P. HILFINGER. A high-level language and silicon compiler for digital signal processing. In *Custom Integrated Circuits Conference*, 1985.

[HOA78] C. HOARE. Communicating sequential processes. *Comminications of the ACM*, August 1978.

[HRVJ95] K. HEIN, A. RUCINSKI, N. VALVERDE, and Y. Jiang. MCM as a VLSI successor in electrical engineering curriculum. In *Proc. of the 4th ATW*, May 1995.

[IEE93] IEEE. *IEEE Standard VHDL Language Reference Manual.* IEEE, New York, NY, 1993. IEEE Standard 1076-1993.

[Inc97] Viewlogic Inc. Homepage: http: //www. viewlogic. com/. Software: Powerview, 1997.

[JC91] W. JAWORSKI and T. CUMMINGS. Programming normalization and optimization: Using infomaps as inspection and programming testing tool. In *Canadian Conference on Electrical and Computer Engineering*, Quebec City, Sept. 1991.

[JIA95] Y. JIANG. MCM design methodologies. Master's thesis, University of New Hampshire, May 1995.

[JM94] W. JAWORSKI and A. MICHAILIDIS. Recovery and enhancement of system patterns : InfoSchemata and InfoMaps. In *Proc. of 3rd ATW*, May 1994.

[KAT82] R. KATZ. A data base approach for managing VLSI design data. In *Proceeding of the 19th Design Automation Conference*, pages 274–282, 1982.

[KM88] D. KU and G. MICHELI. HardwareC - a language for hardware design. Technical Report CSL-TR-90-419, Standford University, 1988.

[KM91] D. KU and G. De Micheli. *Synthesis of ASICs with Hercule and Hebe*, volume High-Level VLSI Synthesis. Kluwer Academic Publishers, 1991.

[KR78] B. KERNIGHAN and D. RITCHIE. *The C Programming Language*. Englewood Cliffs: Prentice-Hall, 1978.

[LSU89] R. LIPSETT, C. SCHAEFER, and C. USSERY. *VHDL: Hardware Description and Design*. Kluwer Academics Publishers, 1989.

[MAN97] R. MANIWA. Focus report: HDL add-in tools. *Integrated System Design*, pages 46–72, Apr. 1997.

[NEW91] R. NEWTON. Design technology challenges in the 1990s. In *Design Technology STAR*, Oct. 1991.

[PAW91] Z. PAWLAK. *Rough Sets. Theoretical Aspects of Reasoning about Data.* Kluwer Academic Publishers, 1991.

[PB91] J. PARDEY and M. BOLTON. Logic synthesis of synchronous parallel controllers. In *International Conference on Computer Design : VLSI in computer and processors*, 1991.

[REI85] W. REISIG. *Petri Nets: an Introduction.* EATCS Monographs on Theorical Computer Science Volume 4. Springer, 1985.

[RYL49] G. RYLE. *The Concept of Mind.* Barnes and Noble Books, N.Y., 1949.

[SCG93] J.-F. SANTUCCI, A.-L. COURBIS, and N. GIAMBIASI. Behavioral testing of digital circuits. *Journal of Microelectronic Systems Integration*, March 1993.

[SOW84] J. SOWA. *Conceptual Structures : Information Processing in Mind and Machine.* ADDISON-WESLEY publishing company, 1984.

[TLW+90] D. THOMAS, E. LAGNESE, R. WALKER, J. NESTOR, J. RAJAN, and R. BLACKBURN. *Algorithmic and Register-Transfer Level Synthesis: The system Architect's Workbench.* Kluwer Academic Publishers, Boston, 1990.

[TM91] D. THOMAS and P. MOORBY. *The Verilog Hardware Description Language.* Kluwer Acadamic Publishers, 1991.

[VCSR94] N. VALVERDE, A.-L. COURBIS, J.-F. SANTUCCI, and A. RUCINSKI. SPECIAL: a specification language for generation of VHDL behavioral descriptions. In *Proc. of The 3d ATW*, May 1994.

[VNG91a] F. VALID, S. NARAYAN, and D. GAJSKI. SpecCharts : A language for system level synthesis. In *Proc. of CHDL*, Marseille, FRANCE, April 1991.

[VNG91b] F. VALID, S. NARAYAN, and D. GAJSKI. System specification and synthesis with the SpecCharts language. In *Proc. of ICCAD*, 1991.

[VNG91c] F. VALID, S. NARAYAN, and D. GAJSKI. Translating system specification to VHDL. In *The European Conference on Design Automation*, Amsterdam, HOLLAND, February 1991.

[ZEI84] B. ZEIGLER. *Multifacetted Modelling and Discrete Event Simulation*. Academic Press Inc., Orlando, Fl, 1984.

Appendix A

SPECIAL

A.1 Semantic of the Graphical Interface

A.1.1 COMPADD Semantic

CC = e1 COMPADD F2

The condition for a transition is *CC = e1 COMPADD F2* and drawn as in Fig. A-1.

The transition condition from state 1 to state 2 is: e1 COMPADD F2. e1 and F2 are expressed as explained in Chapter 6. If the system is in state 1 (in the example) and the composed condition is true the system will switch to state 2. If one of these two statements is not verified, this change of state does not occur. The following algorithm illustrates the representation behavior of an event:

1. the operative part is computed ;
2. the computational variables are affected with their new value ;
3. the conditions of state change are consulted:
   (a) if the condition e1 COMPADD F2 is verified then
      - the time increases to t + δt ;
      - the non-computational global variables are affected with their new value ;
      - the state change is carried out ;
      - the system executes this principle in item - 1 - for the new state.

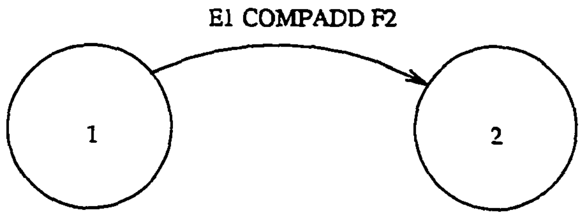


Figure A-1: COMPADD With Event and Fact

(b) if the condition e1 COMPADD F2 is not verified then

- the time increases to t + δt ;
- the non-computational global variables are affected with their new value ;
- the system stays in the same state ;
- the system revalues the condition with the same protocol like item - 3a - but without the revaluation of the non-computational variables.

$$CC = e1\ COMPADD\ e2$$

The condition for a transition is *CC = e1 COMPADD e2* and drawn as in Fig. A-2.

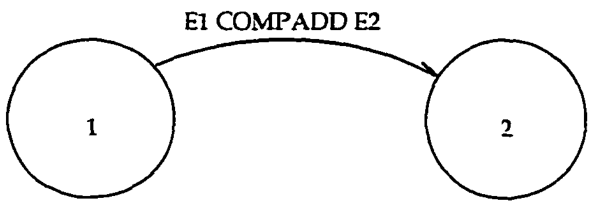


Figure A-2: COMPADD with two events

The transition condition from state 1 to state 2 is: e1 COMPADD e2. e1 and e2 are expressed as explained in Chapter 6. If the system is in state 1 (in the example) and the composed condition is true the system will switch to state 2. If one of these two statements is not verified, this state change is not carried out. The following algorithm illustrates the representation behavior of a composed condition using COMPADD:

1. the operative part is computed ;
2. the computational variables are assigned their new value ;
3. the conditions of state change are consulted:
   (a) if the condition e1 COMPADD e2 is verified then
   - the time increases to t + δt ;
   - the non-computational global variables are assigned their new value ;
   - the state change is carried out ;
   - the system executes this principle in item - 1 - for the new state.

   (b) if the condition e1 COMPADD e2 is not verified then
   - the time increases to t + δt ;
   - the non-computational global variables are assigned their new value ;
   - the system stays in the same state ;
   - the system revalues the condition with the same protocol like item - 3a - but without the revaluation of the non-computational variables.

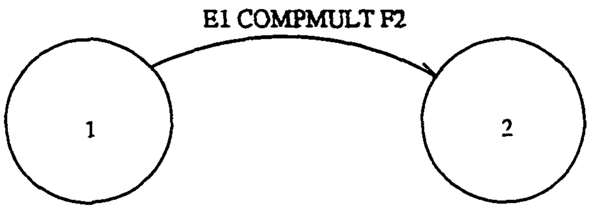


Figure A-3: COMPMULT With Event and Fact

A.1.2 COMPMULT Semantic

CC = e1 COMPMULT F2

The condition for a transition is *CC = e1 COMPMULT F2* and drawn as in Fig. A-3.

The transition condition from state 1 to state 2 is: e1 COMPMULT F2. e1 and F2 are expressed as explained in Chapter 6. If the system is in state 1 (in the example) and the composed condition is true the system will switch to state 2. If one of these two statements is not verified, this change of state does not occur. The following algorithm illustrates the representation behavior of an event:

1. the operative part is computed ;
2. the computational variables are assigned their new value ;
3. the conditions of state change are consulted:
   (a) if the condition e1 COMPMULT F2 is verified then
      - the time increases to t + δt ;
      - the non-computational global variables are assigned their new value ;
      - the state change is carried out ;
      - the system executes this principle in item - 1 - for the new state.

   (b) if the condition e1 COMPMULT F2 is not verified then
      - the time increases to t + δt ;
      - the non-computational global variables are assigned their new value ;
      - the system stays in the same state ;
      - the system revalues the condition with the same protocol like item - 3a - but without the revaluation of the non-computational variables.

CC = F1 COMPMULT F2

The condition for a transition is *CC = F1 COMPMULT F2* and drawn as in Fig. A-4.

The transition condition from state 1 to state 2 is: F1 COMPMULT F2. F1 and F2 are expressed as explained in Chapter 6. If the system is in state 1 (in the example) and the composed condition is true the system will switch to state 2. If one of these two statements is not verified, this change of state is not carried out. The following algorithm illustrates the representation behavior of an event:

1. the operative part is computed ;

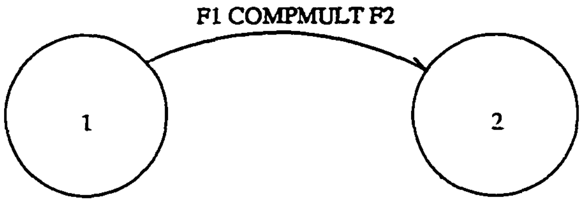


Figure A-4: COMPMULT With Two Facts

2. the computational variables are assigned their new value ;
3. the conditions of state change are consulted:
   (a) if the condition F1 COMPMULT F2 is verified then
      - the time increases to t + δt ;
      - the non-computational global variables are assigned their new value ;
      - the state change is carried out ;
      - the system executes this principle in item - 1 - for the new state.

   (b) if the condition F1 COMPMULT F2 is not verified then
      - the time increases to t + δt ;
      - the non-computational global variables are assigned their new value ;
      - the system stays in the same state ;
      - the system revalues the condition with the same protocol like item - 3a - but without the revaluation of the non-computational variables.

A.1.3 PIPE Semantic

$$CC = F1\ PIPE\ F2$$

The condition for a transition is *CC = F1 PIPE F2* and drawn as in Fig. A-5.

The transition condition from state 1 to state 2 is: F1 PIPE F2. F1 and F2 are expressed as explained in Chapter 6. If the system is in state 1 (in the example) and the composed condition is true the system will switch to state 2. If one of these two statements is not verified, this change of state is not carried out. The following algorithm illustrates the representation behavior of an event:

1. the operative part is computed ;
2. the computational variables are assigned their new value ;
3. the conditions of state change are consulted:
   (a) if the condition F1 is verified then
      - the time increases to t + δt ;
      - the non-computational global variables are assigned their new value ;
      - the condition F2 is verified then
        - the time increases to t + δt ;

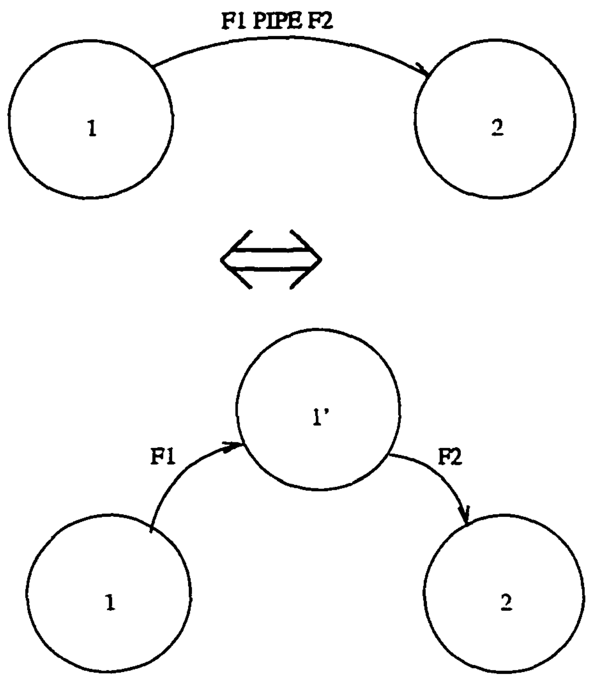


Figure A-5: PIPE with two facts

- the state change is carried out;
- the condition F2 is not verified then
  - the time increases to t + δt ;
  - the system stays in the same state ;

(b) if the condition F1 is not verified then

- the time increases to t + δt ;
- the non-computational global variables are assigned their new value ;
- the system stays in the same state ;
- the system revalues the condition with the same protocol like item - 3a - but without the revaluation of the non-computational variables.

CC = e1 PIPE F2

The condition for a transition is *CC = e1 PIPE F2* and drawn as in Fig. A-6.

The transition condition from state 1 to state 2 is: e1 PIPE F2. e1 and F2 are expressed as explained in Chapter 6. If the system is in state 1 (in the example) and the composed condition is true the system will switch to state 2. If one of these two statements is not verified, this change of state does not occur. The following algorithm illustrates the representation behavior of an event:

1. the operative part is computed ;
2. the computational variables are assigned their new value ;
3. the conditions of state change are consulted:

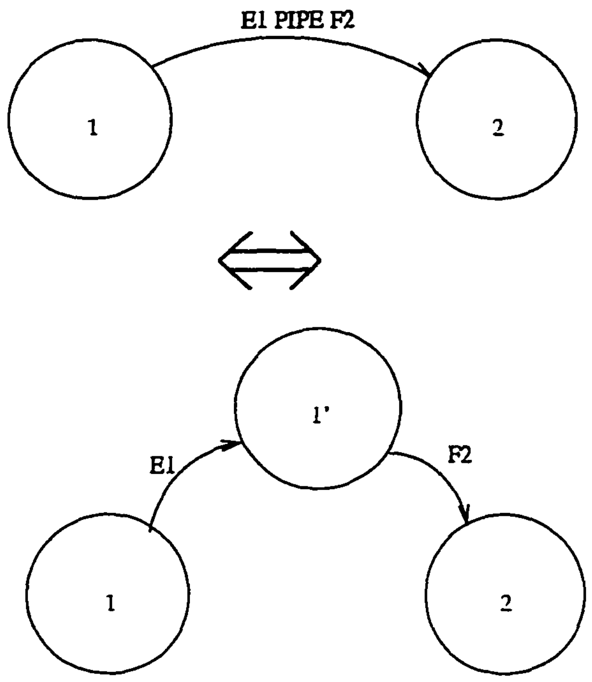


Figure A-6: PIPE With Event and Fact

(a) if the condition e1 is verified then
- the time increases to t + δt ;
- the non-computational global variables are assigned their new value ;
- the condition F2 is verified then
  - the time increases to t + δt ;
  - the state change is carried out;
- the condition F2 is not verified then
  - the time increases to t + δt ;
  - the system stays in the same state ;

(b) if the condition e1 is not verified then
- the time increases to t + δt ;
- the non-computational global variables are assigned their new value ;
- the system stays in the same state ;
- the system revalues the condition with the same protocol like item - 3a - but without the revaluation of the non-computational variables.

CC = F1 PIPE e2

The condition for a transition is *CC = F1 PIPE e2* and drawn as in Fig. A-7.

The transition condition from state 1 to state 2 is: F1 PIPE e2. F1 and e2 are expressed as explained in Chapter 6. If the system is in state 1 (in the example) and the composed condition is true the system will switch to state 2. If one of these two statements is not verified, this change state does not occur. The following algorithm illustrates the representation behavior of an event:

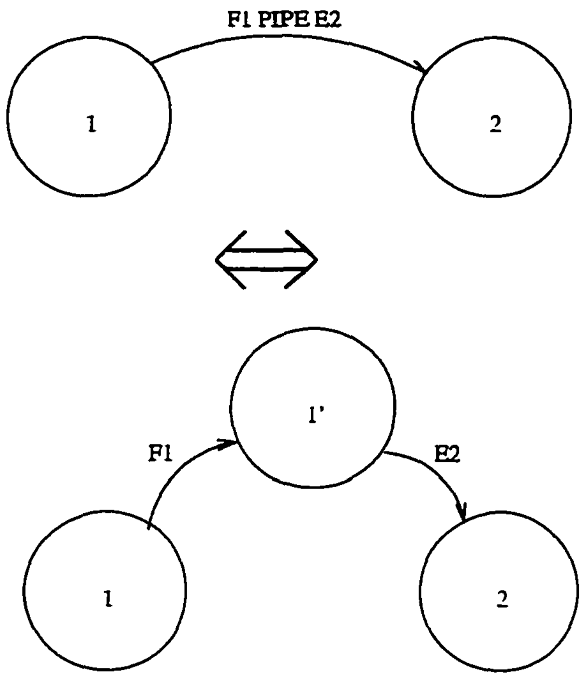


Figure A-7: PIPE with a fact and an event

1. the operative part is computed ;
2. the computational variables are assigned their new value ;
3. the conditions of state change are consulted:
   (a) if the condition F1 is verified then
      - the time increases to t + δt ;
      - the non-computational global variables are assigned their new value ;
      - the condition e2 is verified then
        - the time increases to t + δt ;
        - the state change is carried out;
      - the condition e2 is not verified then
        - the time increases to t + δt ;
        - the system stays in the same state ;

   (b) if the condition F1 is not verified then
      - the time increases to t + δt ;
      - the non-computational global variables are assigned their new value ;
      - the system stays in the same state ;
      - the system revalues the condition with the same protocol like item - 3a - but without the revaluation of the non-computational variables.

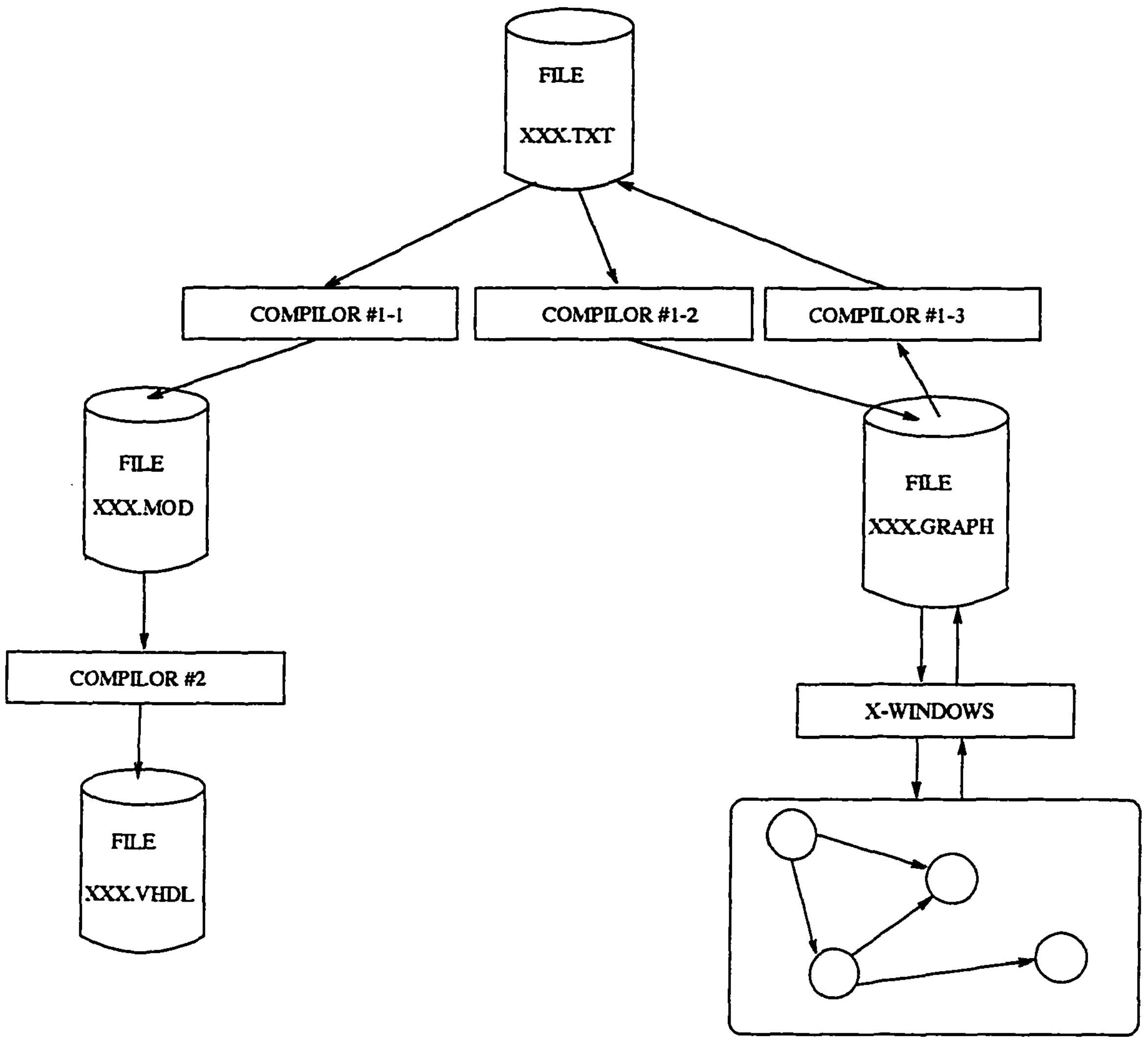


Figure A-8: Software Structure of SPECIAL

A.2 Software Structure

A software was created to implement the function of SPECIAL. It consists of a graphic interface using the X-windows protocol (in particular the xview libraries). Four compilers written LEX and YACC perform the translation of graphical data into a VHDL code file having the extension ".vhdl". A list of intermediate files is generated, which are:

- "XXX.GRAPH" which is the binary form of the graphics;
- "XXX.TXT" which is a direct textual form of the graphics;
- "XXX.MOD" which implements the intermediate model described in chapter 5.

From the file "XXX.TXT", the software can regenerate the file "XXX.GRAPH" which improves the portability of this environment. So, from the graphic interface, a designer can specify a system. By saving the design, the file "XXX.graph" is created. A compiler is then applied to this file to create a "XXX.TXT" file. A second compiler is applied to generate the intermediate representation of the system under design. Finally, a last compiler translates this intermediate model into a VHDL file.

Appendix B

Basic Set Theory

This appendix is a brief overview of the set theory. Section B.1 discusses about the language used in the set theory. Section B.2 introduces the notion of classes over a set.

B.1 The Basic Language of Set Theory

We assume the notion of set. A set E is a term having a relation: $\in$ ($a \in E$ means than a is in E). Intuitively, E is a collection of objects a such as $a \in E$ except all the others.

The language which we shall use for set theory is the first-order predicate calculus with equality. Higher order predicate calculus is an extension of the first-order one. The basic language consists of all the expressions obtained from $x = y$ and $x \in E$ by the sentential connectives $\neq$ (not), $\Rightarrow$ (if ... then ...), $\wedge$ (and), $\vee$ (or), $\Leftrightarrow$ (if and only if), and the quantifiers $\exists x$ (there exists x) and $\forall x$ (for all x). These expressions are called *formulae*. For metamathematical purposes we can consider the connectives $\neq$ and $\vee$ as the only primitive connectives, and the other connectives are considered as obtained from the primitive connectives (i.e. $\phi \wedge \psi$ is $\neg(\neg\phi \vee \neg\psi)$). For the same reason. we can consider $\exists$ as the only primitive quantifier. We also use the abbreviation $x \neq y$ and $x \notin E$ for $\neg x = y$ and $\neg x \in E$. When we write $\exists! x\phi$ we read: there is exactly one x such that ϕ, for the formula $\exists y \forall x(x = y \Leftrightarrow \phi)$ where y is a free variable (i.e. a free variable can have different values). Finally, we can write $(\exists x \in E)\phi$ and $(\forall x \in E)\phi$ for $\exists x(x \in E \vee \phi)$ and $\forall x(x \in E \Rightarrow \phi)$ respectively, and read: "there is an x in E such that ϕ", and "for all x in E, ϕ".

A formula with free variables says something about the value of its free variables. A formula without free variables makes a statement not about the value of some particular variable, but about the universe which the language describes. A formula of the latter kind is called a *sentence*.

Whenever we use a formula with free variables as an axiom or as a theorem we mean to say that the formula holds for all possible values given to its free variables. Thus, if we state a theorem $\exists U(U = V \cup W)$ we mean $\forall W \forall V \exists U(U = V \cup W)$

By a *theory* we mean a set of formulae, which are called *axioms* of the theory. If T is a theory, we write $T \vdash \phi$ for "ϕ is provable from T".

When we refer to a formula as $\phi(x)$ this means that we are interested in the relevant cases where x is a free variable.

B.2 Classes

A class is given by a formula $\phi(x)$ as the class of objects x for which $\phi(x)$ holds. Such a class is denoted $\{x \mid \phi(x)\}$. The expression $\{x \mid \phi(x)\}$ is called a *class term*. The formula may also contain free variables other than x. These other variables are called *parameter*. Different values of the parameters may yield different classes. For example, the class $\{x \mid x \textit{ is a natural number} \wedge x < y\}$ is a class with no member if $y = 0$, has a single number if $y = 1$, and so on. Note also that sets are classes too i.e. the set E is the class $\{x \mid x \in E\}$

Since $\{x \mid \phi(x)\}$ is a class of all x's for which $\phi(x)$ holds, we take the statement $y \in \{x \mid \phi(x)\}$ to stand for $\phi(y)$ (where $\phi(y)$ is the formula obtained from $\phi(x)$ by proper substitution of y for x). Since we consider two sets with the same members to be equal, we should also consider two classes with the same member as equal. We can have the statement $\{x \mid \phi(x)\} = \{y \mid \psi(y)\}$ which stand for $\forall z(\phi(z) \Leftrightarrow \psi(z))$. Consequently, if $y \in \{x \mid \phi(x)\}$ then $x \in \{y \mid \psi(y)\}$ and $\{x \mid \phi(x)\} = \{y \mid \psi(y)\}$. Since the sets are classes, we admit also the statement $E = \{x \mid \phi(x)\}$ and $\{x \mid \phi(x)\} = E$ and let them stand for $\forall z(x \in E \Leftrightarrow \phi(z))$. Saying that one class is a member of the other means that the first class is equal to a set which is member of the other. Accordingly, we admit the statement $\{x \mid \phi(x)\} \in \{y \mid \psi(x)\}$ and let it stand for $\exists z(z = \{x \mid \phi(x)\} \vee z \in \{x \mid \phi(x)\})$, and similarly we let the statement $\{x \mid \phi(x)\} \in y$ stand for $\exists z(z = \{x \mid \phi(x)\} \wedge z \in y)$.

B.3 Relations

Relations, in the set theory, is an important notion. A class S is said to be a (binary) *relation* if every member x of S is an ordered pair. We write them $y\ S\ z$ for $< y, z > \in S$. Moreover, we say that a relation is an *equivalence relation* on a class A when there is a function F on A such that

$$\forall x, y \in A, F(x) = F(y) \Leftrightarrow xRy \tag{B.1}$$

The classes $\{u \mid uRx\}$ are called *equivalent classes* of the relation R. If R is such that its equivalence classes are sets, then we can define $F(x) = \{u \mid uRx\}$ and it is easily seen that Expression B.1 holds. We consider often equivalent classes as sets. The values of $F(x)$ are indeed sets and can be regarded as the representatives of the equivalence classes. Therefore, as introduced in Section B.2, the class $F(x)$ can be a class term of a set A^*. So, we admit that an *equivalence class quotient* of A under R, noted $A^* = A/R$, is defined as

$$A^* = \bigcup_{z \in A} \{u \mid uRz\} \tag{B.2}$$

where $y \in \{u \mid uRz\} \Leftrightarrow \{u \mid uRy\} = \{u \mid uRz\}$.

B.4 Fundamentals of Morphism

In the set theory, a *structure* is an "ordered pair" $< A, R >$ where A is a class and $R \subseteq A \times A$ (R is a binary relation on A). A is said to be the *universe* or the *class* (or the *set*, if appropriate) of the structure $< A, R >$. $< A, R >$ is said to be a structure on the *class* A.

The structure $< B, S >$ is a *substructure* of $< A, R >$ if $B \subseteq A$ and $S = R \mid B$ (i.e. for all $x, y \in B, y \in B$ $xSy \Leftrightarrow xRy$).

A function F is a *morphism* or *homomorphism* of the structure $< A, R >$ *into* the structure $< B, S >$ if F is an injection of A into B and for all $x, y \in A, xRy \Leftrightarrow F(x)SF(y)$. An isomorphism of the structure $< A, R >$ *onto* the structure $< B, S >$ if F is a bijection of A onto B and for all $x, y \in B, xSy \Leftrightarrow F^{-1}(x)RF^{-1}(y)$.

Appendix C

VHDL Code of a RAM Cell

This appendix contains the VHDL code of the RAM cell described in Section 2.3. Most of the code was automatically generated from a graphical description called Design Architect from Mentor Graphics CAD environment. Section C.1 contains the VHDL code from the top level data flow of the RAM cell.

C.1 Top Level VHDL code of the RAM Cell

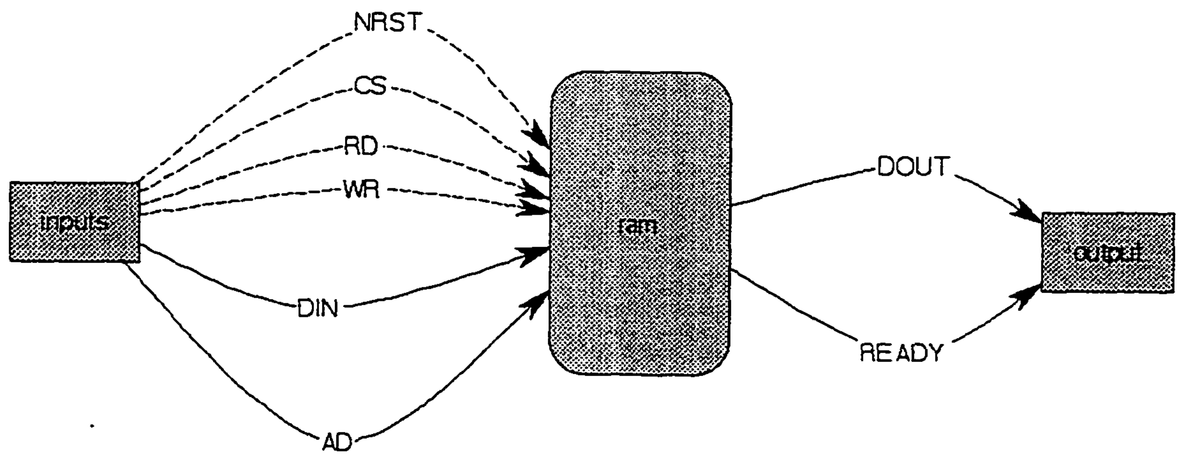


Figure C-1: Context Diagram

This code defines the interface of the RAM cell. It is a description of Fig. C-1. So, the corresponding VHDL code automatically generated by Design Architect is in two parts. The first one is the entity which defines the interface of the RAM cell. In Fig. C-1, the following entity declaration interprets the arrows shown.

```
--
-- Component : ram
--
-- Generated by System Architect version v8.5_2.2 by nav on Feb 27, 97
--
-- Source views :-
-- $DESIGNS/ram/ram_types/types
--
LIBRARY std ;
USE std.standard.all;
LIBRARY designs_ram_sdslocal ;
USE designs_ram_sdslocal.ram_types.all;

ENTITY ram IS
   PORT (
      AD : IN address_type;
      CS : IN bit;
      DIN : IN data_type;
      NRST : IN bit;
      RD : IN bit;
      WR : IN bit;
      DOUT : OUT data_type;
      READY : OUT bit
   );
END ram ;
```

The second part of the VHDL description is the architecture of the description. This description is graphically represented as shown in Fig. C-2

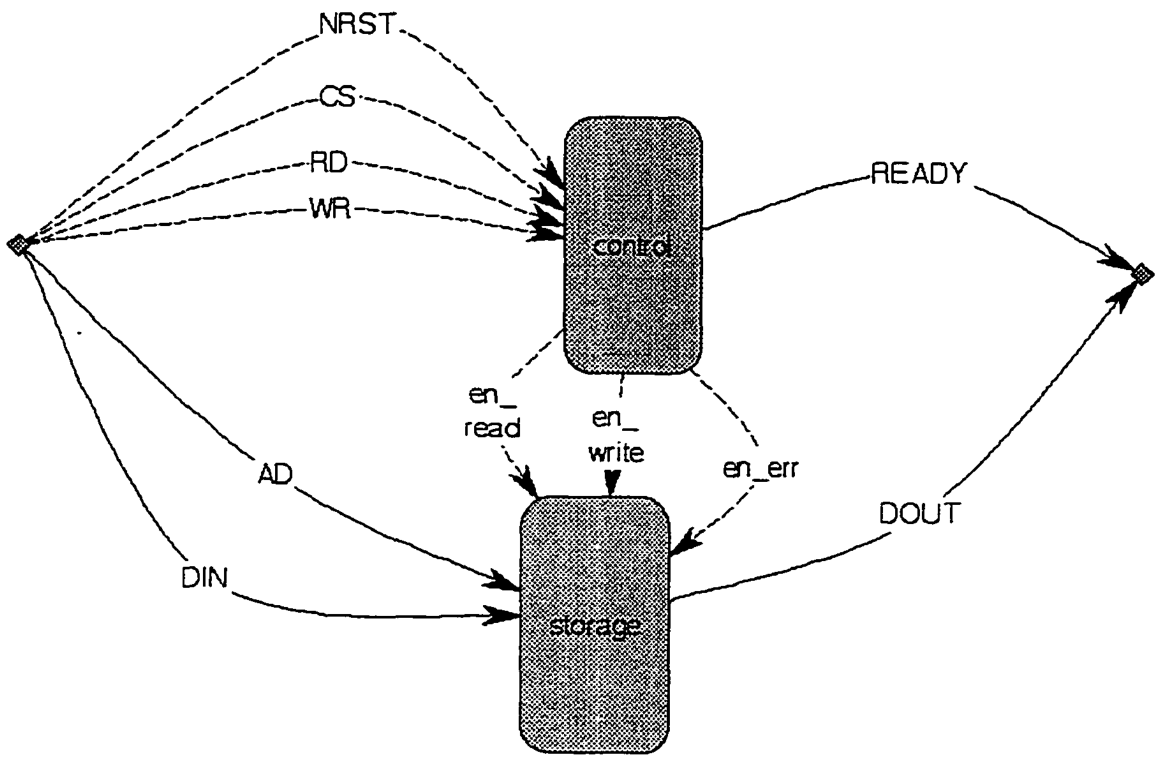


Figure C-2: Dataflow in Mentor Graphics' Design Architect

The VHDl code models a block in the dataflow i.e. "control" and "storage" using the statement component. The interconnections are made through particuliar variables in VHDL called signal.

```
--
-- Component : ram
--
-- Generated by System Architect version v8.5_2.2 by nav on Feb 27, 97
--
-- compatible :: AutoLogic II
-- Source views :-
-- $DESIGNS/ram/data_flow
--

ARCHITECTURE data_flow OF ram IS
   COMPONENT control
      PORT (
         CS : IN bit;
         NRST : IN bit;
         RD : IN bit;
         WR : IN bit;
         en_err : OUT bit;
```

```
        en_read : OUT bit;
        en_write : OUT bit;
        READY : OUT bit
     );
  END COMPONENT ;

  COMPONENT storage
     PORT (
        AD : IN address_type;
        DIN : IN data_type;
        en_err : IN bit;
        en_read : IN bit;
        en_write : IN bit;
        DOUT : OUT data_type
     );
  END COMPONENT ;


  FOR ALL : control USE ENTITY designs_ram_sdslocal.control ;
  FOR ALL : storage USE ENTITY designs_ram_sdslocal.storage ;

  -- Internal Signals
  SIGNAL en_err : bit ;
  SIGNAL en_read : bit ;
  SIGNAL en_write : bit ;

BEGIN

  instance_control : control
     PORT MAP (
        CS,
        NRST,
        RD,
        WR,
        en_err,
        en_read,
        en_write,
        READY
     );

  instance_storage : storage
     PORT MAP (
        AD,
        DIN,
        en_err,
        en_read,
        en_write,
        DOUT
```

```
      );

END data_flow ;
```

Notice that several signals have a none standard type such as bit, bit_vector. In VHDL, the user can customize signal types and it is performed as follows:

```
--
-- Component : ram_types
--
-- Generated by System Architect version v8.5_2.2 by nav on Feb 27, 97
--

PACKAGE ram_types IS
  SUBTYPE data_type IS bit_vector(3 DOWNTO 0) ;
  SUBTYPE address_TYPE IS bit_vector(7 DOWNTO 0) ;
END ram_types ;
```

Therefore, for each block in Fig. C-2, a description needs to be provided. So, in the case of the RAM cell, the block referred to as "control" is a state machine as shown in Fig. C-3.

The corresponding description of Fig. C-3 is decomposed in two parts: the interface definition and the description itself. So, the interface is:

```
--
-- Component : control
--
-- Generated by System Architect version v8.5_2.2 by nav on Feb 27, 97
--
-- Source views :-
-- $DESIGNS/ram/ram_types/types
--
LIBRARY std ;
USE std.standard.all;
LIBRARY designs_ram_sdslocal ;
USE designs_ram_sdslocal.ram_types.all;

ENTITY control IS
   PORT (
      CS : IN bit;
      NRST : IN bit;
      RD : IN bit;
      WR : IN bit;
      en_err : OUT bit;
      en_read : OUT bit;
      en_write : OUT bit;
      READY : OUT bit
   );
END control ;
```

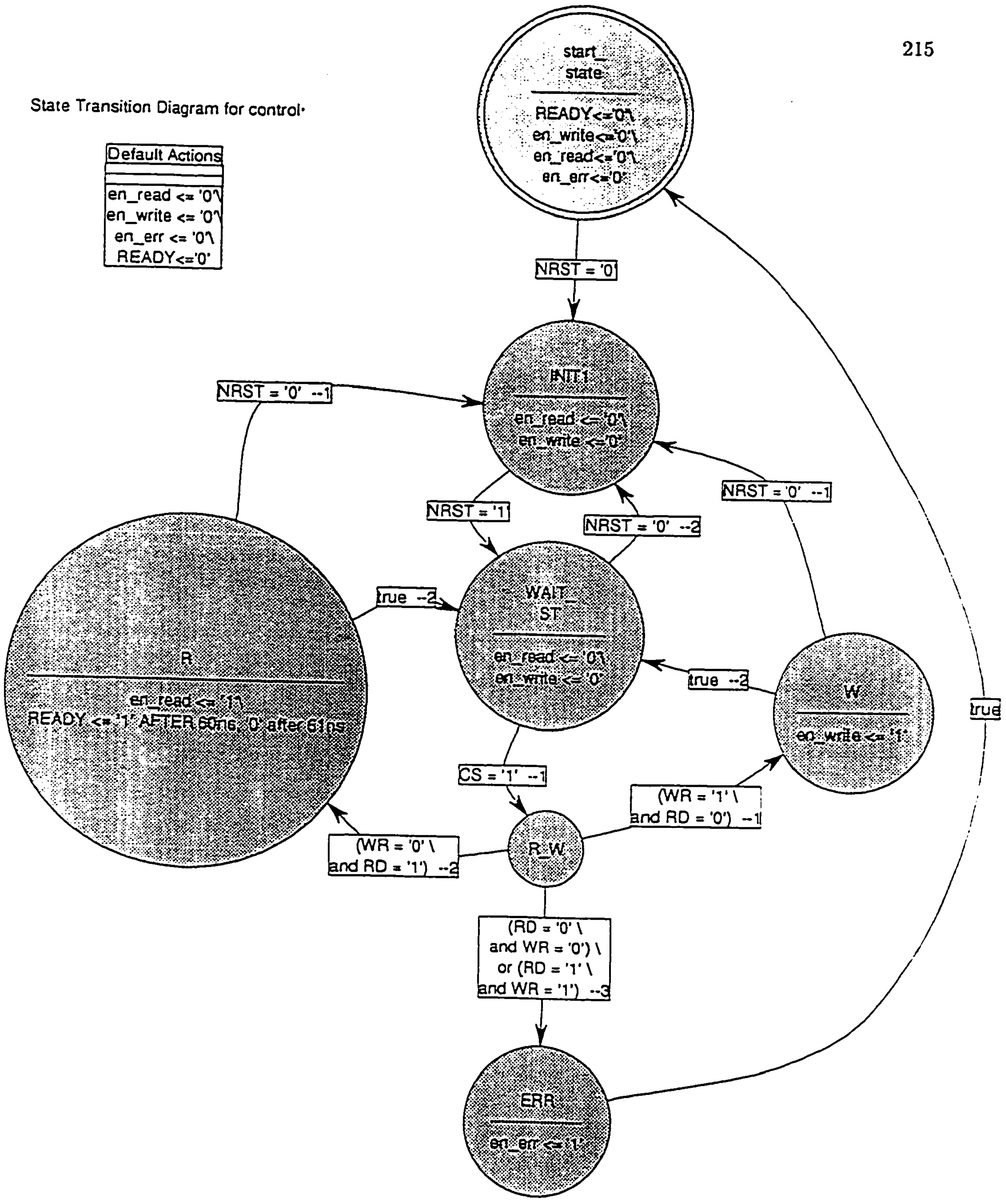


Figure C-3: State machine in Mentor Graphics' System Architect

And, the state machine model is:

```
--
-- Component : control
--
-- Generated by System Architect version v8.5_2.2 by nav on Feb 27, 97
--
-- sensitivity_attr :: 'transaction
-- Source views :-
-- $DESIGNS/ram/control/state_machine
-- $DESIGNS/ram/ram_types/types
--

ARCHITECTURE state_machine OF control IS
   TYPE control_state_type is (
      start_state,
      INIT1,
      WAIT_ST,
      R_W,
      R,
      W,
      ERR
   );


   -- SDS Defined State Signals
   SIGNAL current_state : control_state_type := start_state ;
   SIGNAL next_state : control_state_type := start_state ;
BEGIN

   -------------------------------------------------------------------
   clocked : PROCESS (
      next_state

   )
   -------------------------------------------------------------------
      VARIABLE prop_delay : time := 1 ns ;
   BEGIN
      current_state <= next_state after prop_delay ;
   END PROCESS clocked ;

   -------------------------------------------------------------------
   set_next_state : PROCESS (
      current_state,
      CS'transaction,
      NRST'transaction,
      RD'transaction,
      WR'transaction
)
----------------------------------------------------------------------
BEGIN
   next_state <= current_state;
   CASE current_state IS
   WHEN start_state =>
      IF ( NRST = '0' ) THEN
         next_state <= INIT1;
      END IF;

   WHEN INIT1 =>
      IF ( NRST = '1' ) THEN
         next_state <= WAIT_ST;
      END IF;

   WHEN WAIT_ST =>
      IF ( CS = '1' ) THEN
         next_state <= R_W;
      ELSIF ( NRST = '0' ) THEN
         next_state <= INIT1;
      END IF;

   WHEN R_W =>
      IF ( (WR = '1' and RD = '0') ) THEN
         next_state <= W;
      ELSIF ( (WR = '0' and RD = '1') ) THEN
         next_state <= R;
      ELSIF ( (RD = '0' and WR = '0') or (RD = '1' and WR = '1') ) THEN
         next_state <= ERR;
      END IF;

   WHEN R =>
      IF ( NRST = '0' ) THEN
         next_state <= INIT1;
      ELSIF ( TRUE ) THEN
         next_state <= WAIT_ST;
      END IF;

   WHEN W =>
      IF ( NRST = '0' ) THEN
         next_state <= INIT1;
      ELSIF ( TRUE ) THEN
         next_state <= WAIT_ST;
      END IF;

   WHEN ERR =>
      IF ( TRUE ) THEN
         next_state <= start_state;
      END IF;

   WHEN OTHERS =>
      NULL;
   END CASE;

END PROCESS set_next_state ;

---------------------------------------------------------------------
unclocked : PROCESS (
   current_state,
   CS'transaction,
   NRST'transaction,
   RD'transaction,
   WR'transaction
)
---------------------------------------------------------------------

---------------------------------------------------------------------
BEGIN
   -- Default Actions
   en_read <= '0';
   en_write <= '0';
   en_err <= '0';
   READY<='0';

   -- State Actions
   CASE current_state IS
   WHEN start_state =>
      READY<='0';
      en_write<='0';
      en_read<='0';
      en_err<='0';
   WHEN INIT1 =>
      en_read <= '0';
      en_write <='0';
   WHEN WAIT_ST =>
      en_read <= '0';
      en_write <= '0';
   WHEN R =>
      en_read <= '1';
      READY <= '1' AFTER 60ns, '0' after 61ns;
   WHEN W =>
      en_write <= '1';
   WHEN ERR =>
      en_err <= '1';
   WHEN OTHERS =>
      NULL;
      END CASE;

   END PROCESS unclocked ;
END state_machine ;
```

For the block referred to as "storage" in Fig C-2, the corresponding description is a customized description written directly in VHDL. It specifies a method of storing data. The interface definition is:

```
--
-- Component : storage
--
-- Generated by System Architect version v8.5_2.2 by nav on Feb 27, 97
--
-- Source views :-
-- $DESIGNS/ram/ram_types/types
--
LIBRARY std ;
USE std.standard.all;
LIBRARY designs_ram_sdslocal ;
USE designs_ram_sdslocal.ram_types.all;

ENTITY storage IS
   PORT (
      AD : IN address_type;
      DIN : IN data_type;
      en_err : IN bit;
      en_read : IN bit;
      en_write : IN bit;
      DOUT : OUT data_type
   );
END storage ;
```

And, the model of the storage function is:

```
--
-- Component : storage
--
-- Generated by System Architect version v8.5_2.2 by nav on Feb 27, 97
--
-- sensitivity_attr :: 'transaction

ARCHITECTURE spec OF storage IS
BEGIN

   ------------------------------------------------------------------
  vhdl_storage : PROCESS (
                         AD'transaction,
                         DIN'transaction,
                              en_read'transaction,
                              en_write'transaction,
                              en_err'transaction)
------------------------------------------------------------------
    constant   T_READY_U       : time := 60 ns;
    constant   T_READY_D       : time := 1 ns;
    constant   T_ACCES         : time := 40 ns;
    constant   T_WRITE         : time := 5 ns;

     VARIABLE prop_delay       : TIME := 1 ns;
     CONSTANT nb_words          : integer := 2**8;
     TYPE type_memoire IS ARRAY (0 to nb_words-1) of BIT_VECTOR(0 to 3);
     VARIABLE M                : TYPE_memoire;

     function value(bv : in BIT_VECTOR) return natural is
       variable n : natural := 0;
     begin
       for l in bv'low to bv'high loop
         n := n*2;
         if bv(l) = '1' then
           n:= n+1;
         end if;
       end loop;
       return n;
     end value;
   BEGIN
     IF (en_write = '1') THEN
       M(value(AD)) <= DIN after T_write;
     ELSIF (en_read = '1') THEN
       DOUT <= M(value(AD)) after T_acces;
     ELSIF (en_err = '1') THEN
       Assert FALSE
         report "Wrong Values are observed on WR and RD on the rising edge of CS"
         severity WARNING;
     ELSE
       NULL ;
     END IF;
   END PROCESS vhdl_storage ;
END spec ;
```

IMAGE EVALUATION TEST TARGET (QA-3)