An Expert Assistant for Hardware Systems Specification

Laurent Chaouat, Alain Vachoux, Daniel Mlynek

Swiss Federal Institute of Technology (EPFL)
EE Dpt., Integrated Systems Center (C3i)
CH-1015 Lausanne, Switzerland
E-MAIL: laurent.chaouat@leg.de.epfl.ch

Abstract: *This paper presents the Module Manager, which is a novel approach to assist the designer in the specification of hardware systems. This flexible expert system proposes behavioural solutions, at a high level of abstraction, that match the designer's requirements. Models are selected from a repository of designs previously specified within the MODES environment. Thereby, the Module Manager allows them to be reused, hence ensuring their generic nature. This paper focuses on the architecture and the mechanisms of the Module Manager.*

1 Introduction

Market competitiveness demands that the designer use an efficient methodology [1] and has a good technical background to master the increasing number and diversities of designs. Furthermore, due to the ever growing complexity of hardware systems, the designer is often confronted with the dilemma of seeking a trade-off between efficiency, rapidity, quality and cost. The solution to a particular problem is far from easy and exclusive. In order to reduce some of these difficulties, sophisticated CAD tools have contributed significantly to producing solutions that accord with the customer's needs. Meanwhile, it remains difficult to generate correct solutions of good quality. In fact, designers need a flexible tool that can propose a panorama of solutions for different domains, such as microprocessors, DSP, microsystems, communication protocols and many others.

The Module Manager is an expert system used as a prototyping approach to ease the process of designing a hardware system. The main objective of such a tool is to reduce the time and the modelling expertise needed to generate behavioural models. It aims to assist the designer by providing a knowledge base to generate a set of behavioural models corresponding to the requirements definition. The selected solutions are represented graphically using existing tools (e.g.: speedCHART™ [18], Visual HDL™ from SEE Technologies) or in a more textual manner (VHDL). This objective is achieved by guiding the designer, allowing him to describe devices incrementally in a very abstract manner, somewhat like a high level datasheet description. The expert system can also be used to train the user in hardware modelling by explaining its reasoning process.

The Module Manager is involved during the specification phase of a hardware system. Using the requirements specification as a starting point, it is able to search in a repository of previously specified models for a set of possible solutions that could be suitable for the hardware system the designer is working on. However, the suggested models may not be directly appropriate in a first stage. The models should then be manually modified using graphical or textual editors.

This is a practical approach that gives a quick overview and a better idea of the different parts of a system to design. It also avoids reinventing the wheel by creating new behavioural models from scratch. The Module Manager is part of the MODES (**MOD**eling **E**xpert **S**ystem) project [4] [5].

Related work

Hitherto, most related works have focused on providing either efficient knowledge-based systems for a specific application domain at a low level of abstraction, or design management assistance for a particular VLSI task. For example, SISC [15] is a frame-based system, customised to represent knowledge about integrated circuits. Kinden [14] is an experimental knowledge-based intelligent environment for the VLSI design process. Micon1 [11] is a synthesis tool that aims to automate the design of computer systems. DEBYS (Design BY Specification) [12] covers both operative and technological specifications for digital and analog systems. DEBYS is mainly used for the design of microsystems.

Our research differs from these efforts by proposing an intelligent and flexible architecture able to manage the reusability of behavioural models for the specification of new designs at a high level of abstraction.

About this paper

In the present paper we first introduce the MODES environment and its interaction with the Module Manager. We also give a general overview of the CSIF format, a textual representation which preserves the specification structure of different formalisms. Since the CSIF format is an important aspect of the Module Manager, in Section 3 we present the mechanisms of the Module Manager that interact with CSIF. In Section 4, we discuss the global concept of the Module Manager. Section 5 focuses on the knowledge representation of the Module Manager, an important issue for providing a global control over a design. Finally, we present our conclusions.

2 The Module Manager and the MODES environment.

MODES is an environment for specifying electronic devices using high level behavioural formalisms. This section presents the utility working with such an environment and the interaction with the Module Manager in MODES through the CSIF format. A brief review of the CSIF format is given using a simple example.

2.1 The MODES concept.

Due to the complexity of electronic devices such as integrated circuits, application specific integrated circuits (ASIC) or printed board, the designer is forced to follow a top-down approach to correctly achieve his (or her) design within the shortest time. An efficient design should at the earliest stage possible take account of the constraints implied by the environment in which the system will work.

MODES is mainly involved with the system level design [3] where a lot of work is performed for the entire system at a high level of abstraction. It implies the following tasks: (i) specification, (ii) modelling, (iii) partitioning and (iv) integration of environment constraints. The elaboration of behavioural models of hardware systems is at the heart of the system level design. Hardware Description Languages (HDL) have been developed to describe different views of a system, usually the behavioural and structural views, at different levels of abstraction, from the switch

level to the algorithmic level [2]. However, modelling hardware systems with an HDL requires that the designer have a good understanding of hardware systems and a good software programming background, especially for representing the requirements in an HDL code. Consequently, there is a need to provide software tools which should perform the different tasks involved in the system level design. At the present time, several commercial or academic products (e.g.: speedCHART, Visual HDL, KBs, EASE/VHDL, ExpressV-HDL from i-Logix) are available to provide the designer with a variety of editors that capture specifications related to a hardware system in order to automatically generate behavioural HDL models for simulation or synthesis. These tools follow the framework presented in figure 1.

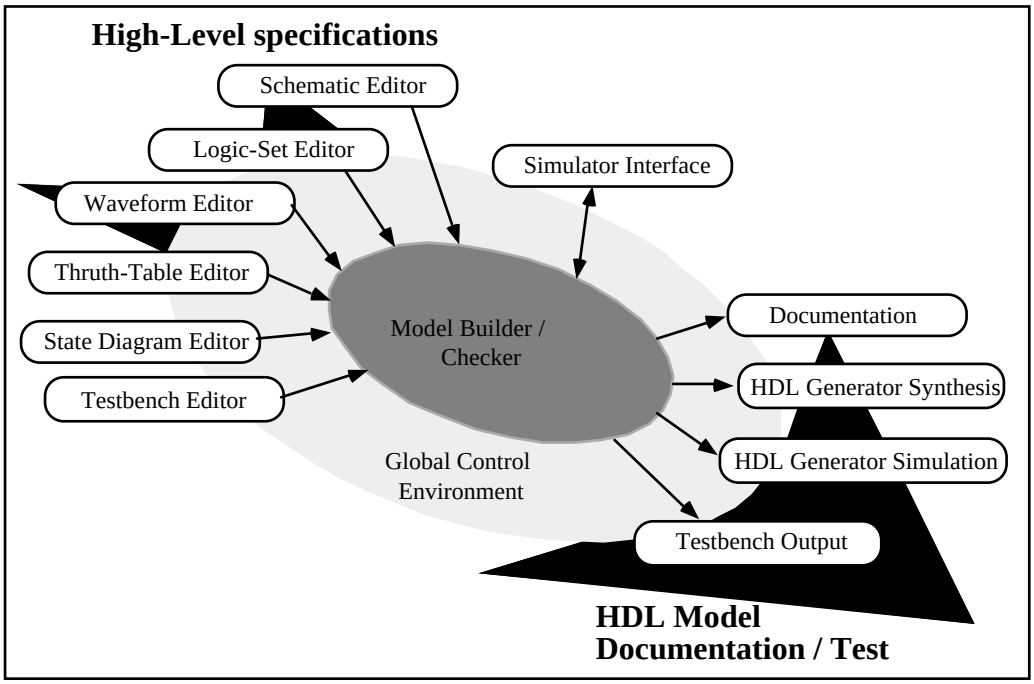


Fig.1: Design environment.

The Module Manager is a software component that can be connected to such a type of architecture in order to enhance its functionality and introduce an expert aspect. By combining the Module Manager with the specification tools, MODES provides the designer with expert assistance for the specification of hardware systems. MODES is organised around three sets of functionalities (figure 2): (i) the graphical capture tools for high-level specification formalisms, (ii) the merge of all the specifications into the **C**ommon **S**pecification **I**ntermediate **F**ormat (CSIF) [6] by the model builder, and (iii) the generation of HDL models for simulation and synthesis. MODES uses a specific knowledge base which constitutes the repository of all the information available (i.e.: modelling guidelines, verification rules and previously instantiated designs). The Module Manager is the component that handles this knowledge base. All the models of the database are stored within the CSIF representation.

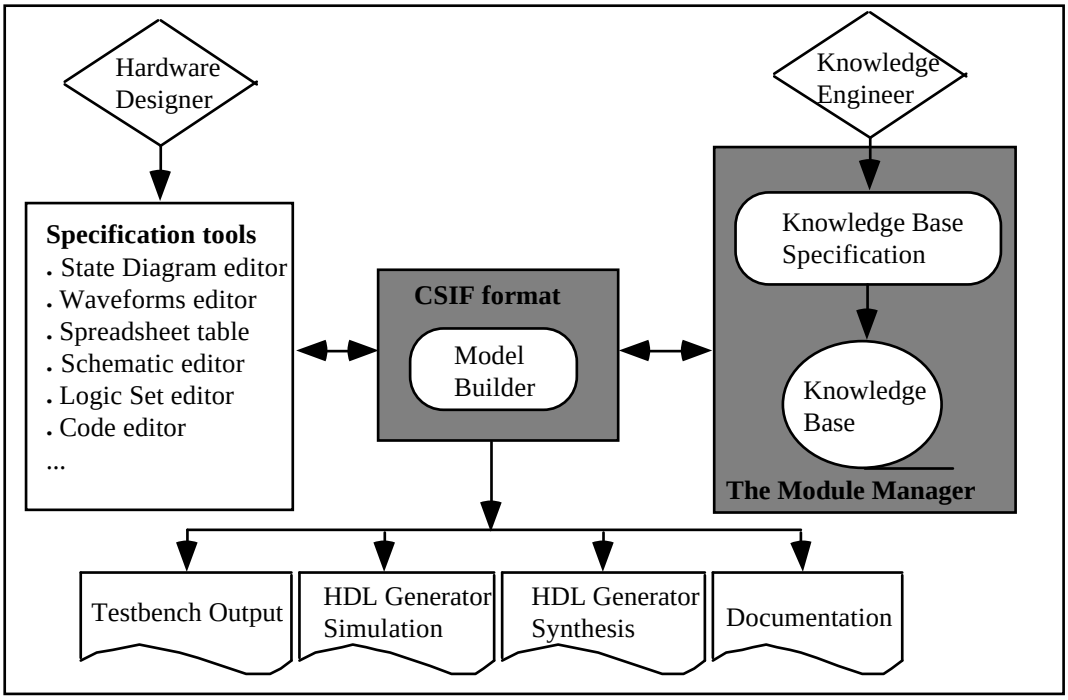


Fig. 2: The MODES Block Diagram.

2.2 A practical solution for merging hardware system specifications: the CSIF format

Proposing an environment with various formalisms gives designer the ability to select the most convenient representation to specify the whole or part of a design. However, the validation process inside each specific editor is not sufficient to check the consistency of the global model. Neither is it easy to allow future extensions, such as adding new editors or applications that may directly use the specifications to extract global properties, nor to evaluate functional performances or consistency.

The CSIF format, based on a textual language, aims to go beyond these constraints. Moreover it respects the way the designer has entered the specifications (i.e.: hierarchy, concurrence, partitioning, formalism, etc.). In this way, the Module Manager can re-create the CSIF specifications in their initial formalism through the appropriate capture tool; something we cannot do with VHDL (figure 3). Since CSIF is designed according to an object-oriented approach, it offers the capability to easily modify some specific aspects of a model, such as bus width or data size, and hence, maintains the genericity of a model. CSIF also offers various mechanisms to enhance its manageability. CSIF provides an efficient communication between tools linked together in a CAD framework and eases their implementation by restricting data transformations from one application to another. An advantage of using such a format is that all the tools work using the same representation.

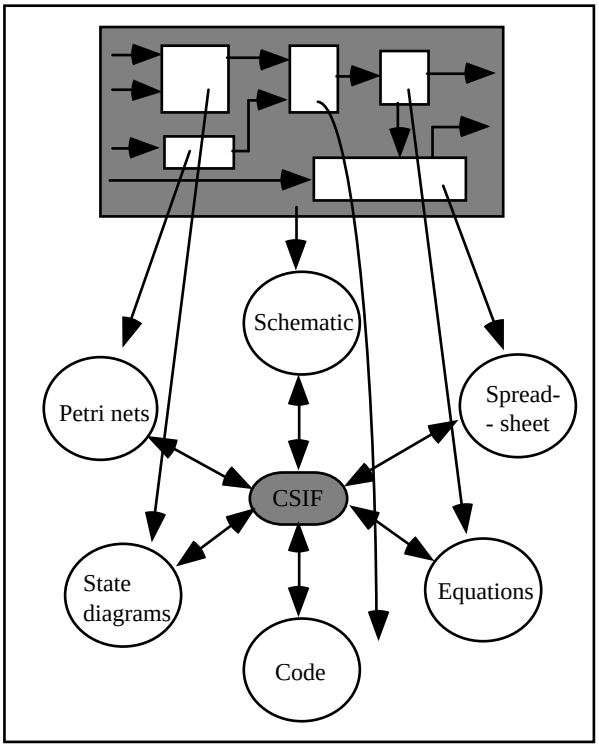


Fig. 3: Merging the specifications into CSIF.

Modules and *Netlist* interconnections are the fundamental elements of this format. A system contains a set of modules describing the behaviour of its functional blocks. A *module* is composed of global declarations defining its external port, global variables used in mapping the behaviour, and a body. CSIF supports three types of decomposition that may be included in a module body: hierarchy, concurrence and behavioural units.

Hierarchy is an important aspect for organising the specifications into different levels of refinement. In CSIF, a hierarchy has the same meaning as in most graphical/textual formalisms used for system specification. A hierarchy can refer either to an *explicit* or to an *implicit diagram.* A diagram maps directly onto the graphical representation inside an editor. An *explicit diagram* represents a hierarchy inside a module as an object. It is a block of actions representing a part of the current module behaviour. Since it is considered as an object, an *explicit diagram* can be invoked in different places in the module behaviour with a calling statement. An *implicit diagram* has the same effect as an *explicit diagram* but is directly involved inside the module behaviour. Therefore, no calling statement can be used to invoke this kind of diagram somewhere else.

Concurrence represents processes that are executed simultaneously. Each concurrent element is independent. However, data can be exchanged between each process. From the editor's side, concurrence is represented in a hierarchy. In the CSIF language concurrent elements are placed inside a diagram.

A *behavioural unit* aims to gather actions and declarations together. It allows the partitioning of the specifications and can be employed for a co-design approach. A behavioural unit is composed of all the elements used to describe a module body: sequential actions, diagrams and other encapsulated behavioural units.

Since diagrams give access to a lower hierarchical level, we can attach to them an *entry action* for the initialisation and an *exit action.* Action blocks can be executed under conditional statements

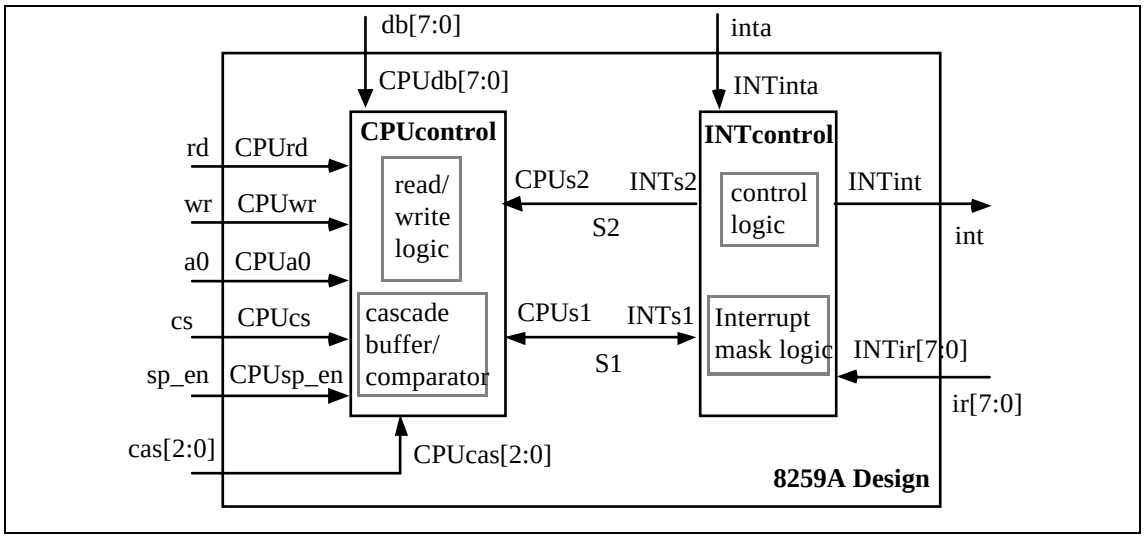
(i.e.: *If-Then-Else, While loop, For loop, Switch*) as a simple set of sequential actions, or recursively as a *behavioural unit* or a *diagram*. This aspect allows the designer to enter different hierarchical levels under certain conditions.

The *Switch* statement is one type of action that we also have to take into account. There are two possible ways to interpret this. Firstly, it can be used as a simple action that behaves the same way as in standard programming. The second way is more related to hardware system behaviour, following an extended state machine model. This approach refers to the BEMCharts language [10]. CSIF provides a control variable to re-create the corresponding control aspect of an FSM (Finite State Machine). The *StateIf* type is used to define variables that control the evolution of an FSM. A *StateIf* variable involved in a *Switch* statement represents the control aspect of an FSM. A state of an FSM (*Case* Statement) may contain four actions: *Entry*, *State action*, *exit* and one or more *transitions*. A state action block will be executed as long as the state is active.

Interconnections of components are performed with a netlist description. The communication between the elements is carried out through port definitions enclosed in the declarative part of the statement. We should also mention that a component can be a *netlist configuration* or a *module configuration*.

Procedures and *functions* can also be used in CSIF to perform specific sequential operations.

We now use a simple example to illustrate the utility using such a format. This example, taken from [6], shows how to use CSIF to represent the various formalisms that we may use to specify a system. We specify the programmable interrupt controller INTEL 8259A [19], which is currently functioning within the 80xx family. This component handles up to eight interrupts in a single mode and up to 64 interrupts in the cascade mode. The 8259A is organised around two main functionalities. The bus interface controller (**CPUcontrol**), managing the configuration of the device, and the communication protocol unit, handling exchanges of information with the serving processor (**INTcontrol**). Figure 4 shows how the 8259A can easily be represented in a schematic form.



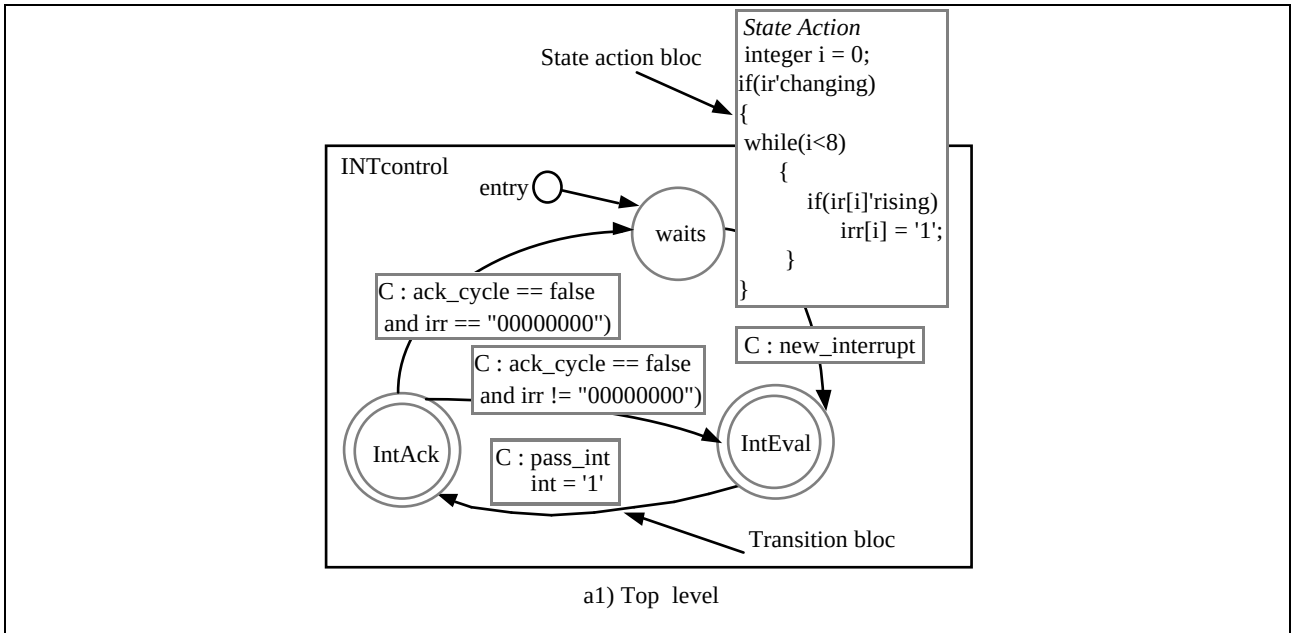
```
NetList PIC_8259A EDITOR : SCE
{
--Interface description
   logic rd     in;
   logic wr     in;
   logic a0     in;
   logic cs     in;
   logic cas[3] in;
   logic db[8]  inout;
   logic sp_en  in;
   logic ir[8]  in;
   logic int    out;
   logic inta   in;
   Signal    logic S2;
   Signal    logic S1[8];
```

```
   CPUcontrol: MODULE EDITOR:STD
   {
      CPUdb => db;
      CPUrd => rd;
      CPUwr => wr;
      CPUcs => cs;
      CPUcas => cas;
      CPUa0 => a0;
      CPUsp_en => sp_en;
      CPUs1 => S1;
      CPUs2 => S2;
   }
   INTcontrol : MODULE
EDITOR:WFE
   {
      INTinta => inta;
      INTint => int;
      INTirc => ir;
      INTs1 => S1;
      INTs2 => S2;
   }
}
```

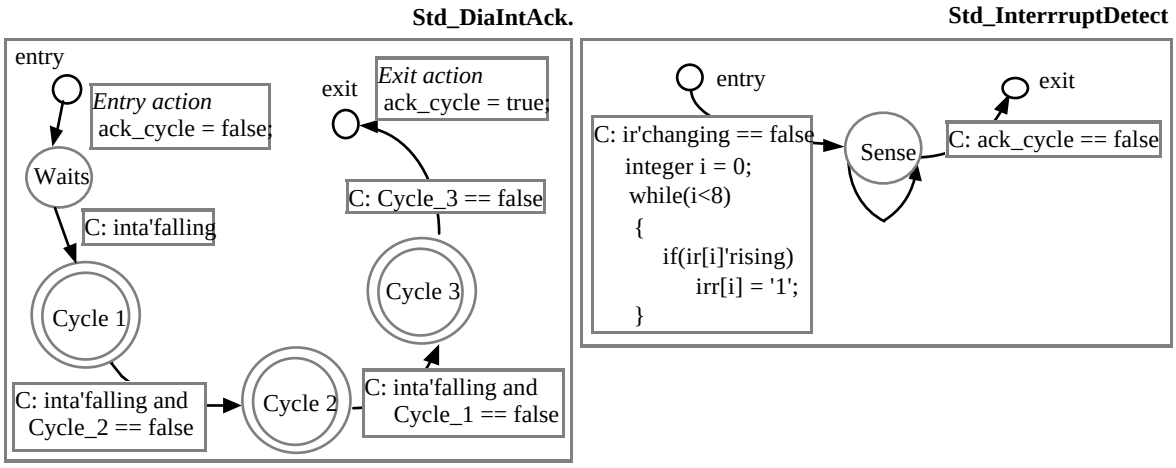
Fig. 4: The INTEL 8259A netlist description.

Communication between both modules, **CPUcontrol** and **INTcontrol**, connected at the same level, is possible through the internal signals **s1** and **s2**. Port declarations are used to connect the corresponding component ports to the upper component interface.

We present a detailed description of both modules in figure 5, and figure 6. The specification of the **INTcontrol** part follows an extended state machine model, composed of three hierarchies: **INTcontrol** as the top level and two lower level hierarchies **IntAck** and **IntEval** under control of two states. Each of these hierarchies is gathered in a *Diagram*. In the CSIF representation, each state machine is represented by a *Switch* statement. The *StateIf* variable **ControlST** involved in the *Switch* statement of the **INTcontrol** module represents the control aspect of its corresponding state machine. A state action is executed as long as the state is active. Therefore, when the **ControlST** variable is evaluated to the '**IntAck**' value, the corresponding state is activated and at the same time enables all the lower states in the hierarchy (**Std_InterruptDetect** and **Std_InterruptAck,** which are concurrent elements). Similarly, changing the state to '**IntEval**' will disable the diagram **Std_DiaIntEval**.



a1) Top level

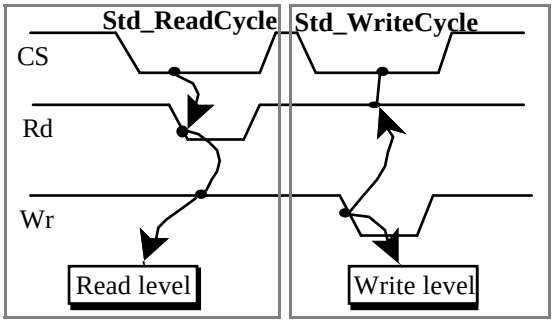


a2) Diagram DiaIntAck

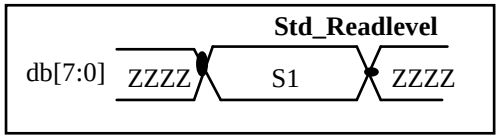
```
MODULE INTcontrol
{
-- Module Interface description
  logic      INTinta      in;
  logic      INTs2        out;
  logic      INTs1[8]     inout;
  logic      INTir[8]     in;
  logic      int          out;

-- Module Global Variables
  boolean    pass_in, ack_cycle;

   {-- Module unit
    INTcontrol:(-- Top-Diagram
description
    {

-- Declarations  of the INTcontrol
FSM
     stateIF ControlST = "Entry";
     integer i_waits;
     boolean new_interrupt;

     SWITCH(ControlST)
     {
      case "Entry" DROPTHROUGH :
        Trans  :
        {ControlST = "waits" ;--
Next state}
      case "waits" :
         StateAction : { -- state
actions bloc}
         Trans  : { -- Next state
evaluation }
      case "IntEval" :
         StateAction :
         {-- Access to
Std_DiaIntEval diagram
          DIAGRAM :Std_DiaIntEval;}
          Trans :  { -- Next state
evaluation }
      case "IntAck" :
         EntryAction :
         {
          ack_cycle  == true;
         }
         StateAction :
         {
         Std_DiaIntAck:(-- diagram
Std_DiaIntAck
         {
          Std_InterruptAck:(
          -- Std_InterruptAck  descr.
          )
         },  -- Concurrent diagrams
         {
         Std_InterruptDetect:(
           -- Std_InterruptDetect descr.
          )
       }
      )  -- end diagram Std_DiaIntAck.
      }
      Trans :  { -- Next state evaluation }
           }
      }
    )
  }

  DIAGRAM Std_DiaIntEval
  {
  -- Std_DiaIntEval external diagram
description
   }
}
```

Fig. 5: Representation of the INTcontrol module.

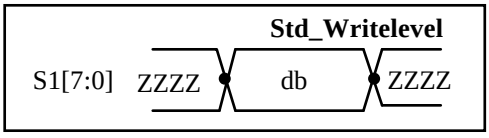
The description of the **CPUcontrol** module follows the same construction, using two state machines, but comes from a waveform specification. At the top level, the **ReadLevel** and the **WriteLevel** are two diagrams giving access to a lower level of the hierarchy constituted by the **Std_ReadCycle** and **Std_WriteCycle** sub-diagrams. These two diagrams are under control of the *StateIf* variables **St_Read1** and **St_Write1**. The next section presents another way to translate such a specification into CSIF.



a1) Top level



a2) ReadLevel diagram



a3) WriteLevel diagram

```
MODULE CPUcontrol
{-- Module interface description
  logic CPUdb[8] inout;
  logic CPUrd    in;
  logic CPUwr    in;
  logic CPUcs    in;
  logic CPUcas   in;
  logic CPUsp_en in;
  logic CPUs1[8] inout;
  logic CPUs2    in;
  {-- Module unit
    CPUcontrol: ( -- top-diagram description
    {-- Declarations of control variables
        stateFI St_Read1 = "Entry";
        stateIF St_Write1 = "Entry";
        ( --State Diagram Std_ReadCycle
         {
          SWITCH(St_Read1) {
            case "Entry" DROPTHROUGH :
               Trans :
               {
                IF(CPUrd'falling AND CPUcs == '0'
                                 AND CPUwr == '1')
                   { St_Read1 = "ReadLevel"; }
                }
            case "ReadLevel" :
               StateAction: {DIAGRAM:
Std_ReadLevel;}
                Trans : { St_Read1= "Entry"; }
          }
        },   -- Concurency diagram indication

        { --State Diagram Std_WriteCycle
          SWITCH(St_Write1) {
            case "Entry" DROPTHROUGH :
               Trans :
                {
                 IF(CPUrd'falling AND CPUcs == '0'
                                  AND CPUwr == '1')
                   { St_Write1 = "WriteLevel"; }
                }
             case "WriteLevel" :
              StateAction: {DIAGRAM:
Std_WriteLevel;}
                Trans : { St_Write1= "Entry"; }
            }
          )
       }
      }
    )   -- End top diagram
    DIAGRAM Std_ReadLevel
    {   CPUdb = CPUs1;   }
    DIAGRAM Std_WriteLevel
    {   CPUs1= CPUdb;  }
  }
}
```

Fig. 6: Representation of the CPUcontrol module.

We have demonstrated the feasibility and the efficiency of this format by implementing a bridge between the specifications coming from the speedCHART tool (state diagrams, spreadsheet, code and netlists) and CSIF.

3 The Module Manager and the CSIF format

The Module Manager interacts in an intelligent way with CSIF by identifying and extracting the specific pieces of behaviour to be reused. The storage of a design follows a predefined scheme which specifies the different types of models and functions, with their relations. This classification includes a hierarchical decomposition, properties of the behaviours, modelling and verification rules (figure 7). A Knowledge Engineer is responsible for the configuration, maintenance and upgrade of the expert system.

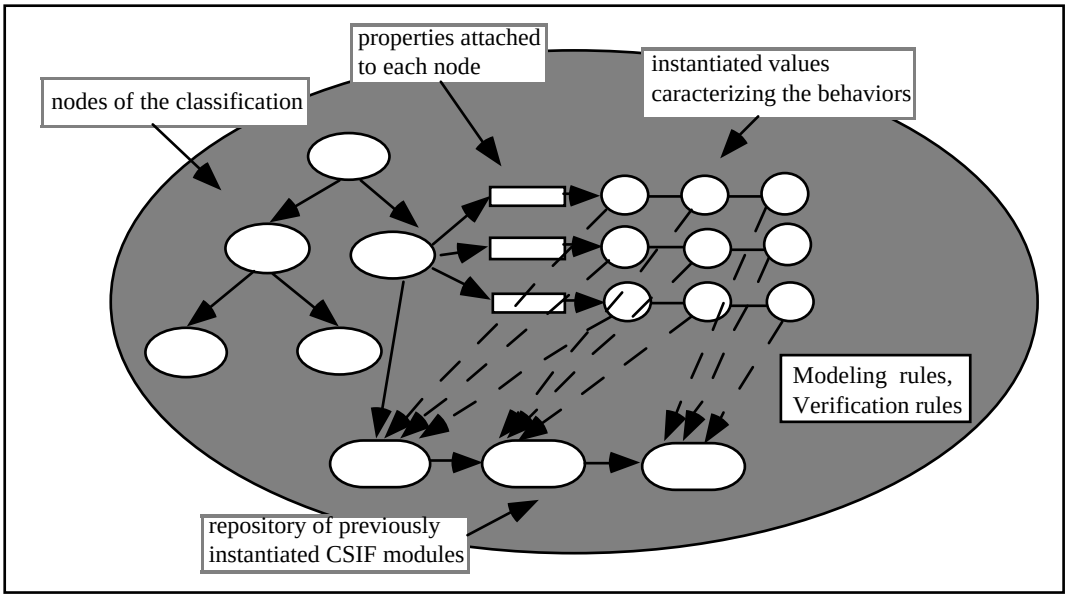


Fig. 7: General structure of the knowledge base.

Each node of the classification contains a variety of information: the relations with other nodes, different properties characterising a node, and the available CSIF instance modules. Properties give some hints to the designer about the different characteristics of the models saved upon each node of the classification.

Figure 8 gives a partial example of a kind of classification for processor architectures. The CSIF specifications are partitioned in such a way that each part refers to a node of the classification. The reusability of a specific module consists of restoring its graphical form (such as waveforms, state diagrams, spreadsheet table), in the appropriate editor tso that it can easily be modified for the specification of new systems. This reusability is related to several types of representation: the whole design, modules, diagrams, netlists and subprograms.

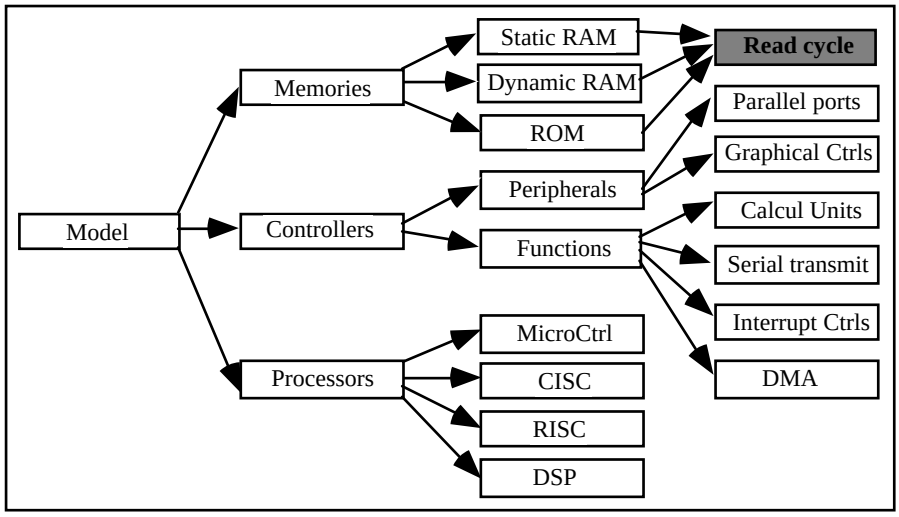


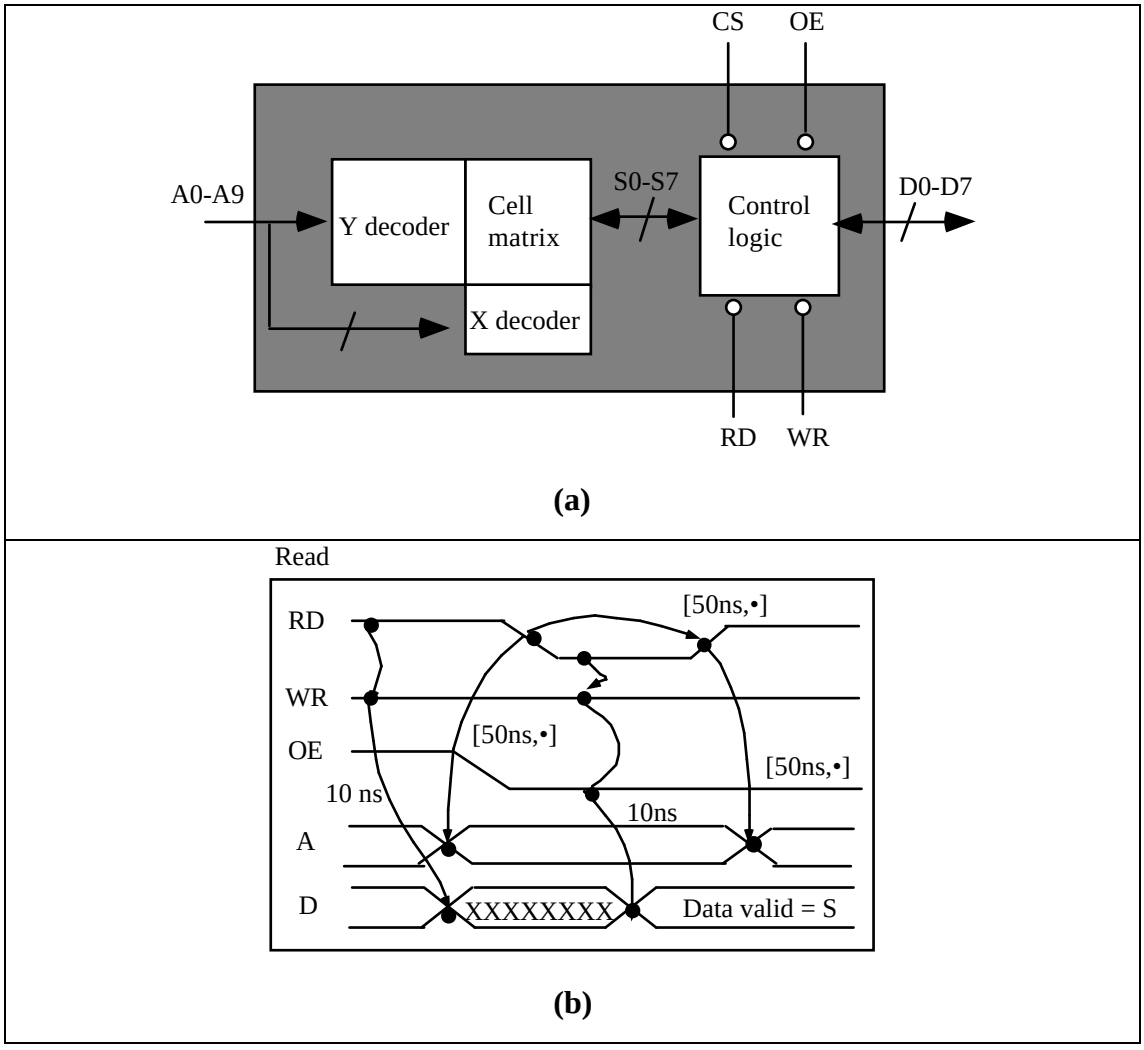
Fig. 8: An example of classification (partial).

When all the specifications are merged into the CSIF form, a first step is to identify the behaviours that will be saved in the database. Our solution is to use a label to name each of the important parts of behaviour. These labels, which are entered by the designer using the capture tools, convey the name of the behaviour as well as the node of the classification tree where it has to be connected.

Figure 9 presents a simple example to illustrate this mechanism. The **control logic** unit of a basic memory (a) has been specified with the waveform editor. Its CSIF representation is given in (c). This representation differs from the one we presented in figure 6. It is essentially based on the predefined function **Check_backward()**, automatically generated by the **E**xtended **T**iming **D**iagram editor (ETD). In this *module*, we are interested in reusing the *diagram* **Read** (b). In order to identify and extract this behaviour, we use the specific label **Read@Read_cycle** where **Read_cycle** is the name of the node where the diagram **Read** has to be classified. The process of storing the corresponding structure in the database is achieved as soon as the properties of the node **Read_cycle** are instantiated.

The fact that CSIF is not combined with any graphical structure enhances its flexibility. However, when reusing a piece of design, the Module Manager should re-create its graphical aspect. Each reusable behaviour is associated with the corresponding graphical representation, which is also saved in the database (figure 8).

We also need to restore all the context in which the behaviour is involved. To do so, a specific mechanism will dynamically search through the whole CSIF structure for all the definitions of variables, signals, types, subprograms and diagrams implied in the behaviour in order to generate a black box, that is directly usable by the end-user through the capture tools.



```
Module Control_logic EDITOR:WFE
{
  -- Global declarations
  -- (external port and variables)
  {
    Top: (
      -- Definition of Top
      DIAGRAM: Control --Explicit
diagram call
    )
  }
  --Explicit Diagram Control
  DIAGRAM Control
  {
   -- Read and Write are concurrent
elements
   (
    {
     -- Implicit diagram Read to be
reused
     -- Use of the specific label
     Read@Read_cycle: (
       if (RD=='1' AND WR=='1')
       {
        D<="ZZZZZZZZ" after 10ns;
        S<="ZZZZZZZZ" after 10ns;
       }
```

```
       else if (RD=='0' AND WR=='1' AND
OE=='0')
       { Check_backward(A'changing,
RD'falling,50ns,
Time'High,Warning,
                        "Backward check
RD->A");
         Check_backward(A'changing,
                        RD'rising,50ns,
Time'High,Warning,
                       "forward check
RD->A");
         Check_backward(RD'falling,
                        RD'rising,50ns,
Time'High,Warning,
                        "forward check
RD->RD");
         D<=S after 10ns;
       }
    ) -- End of Read_cycle description
   },
  {
   -- Implicit diagram Write
   Write: (
     -- Definition of the write cycle
      )
     }
    )
  }
}
```

(b)

Fig. 9: a) A basic RAM architecture,
b) Specification of the Read cycle using the waveform editor.
c) CSIF representation of the Control logic module,

4 The Module Manager architecture

The Module Manager is a software architecture based on a Database Management System (DBMS) written in C++ and on an expert module (knowledge base and inference engine) capable of proposing different possible scenarios of solutions for a system design. This second part is implemented within the Nexpert Object™ environment [17]. The Module Manager is more than a model storage/retrieve system since it is composed of a repository of all information required to facilitate the design process of a system. In this context, it also manages the different versions of a model. Modelling mechanisms offer the necessary information to create new behavioural models by reusing previously instantiated designs.

On the one hand, the Module Manager provides a simple/save restore capability, allowing the designer to complete his design in multiple sessions. On the other, it gives access to the whole or parts of previously completed designs to allow their extensive reuse.

An appropriate frame-based script language called MAGMA (**M**odelling **A**ided **G**uide for **M**ODES **A**pplications) is used for the description of the scheme of the knowledge base and the kinds of properties describing each node of the classification.

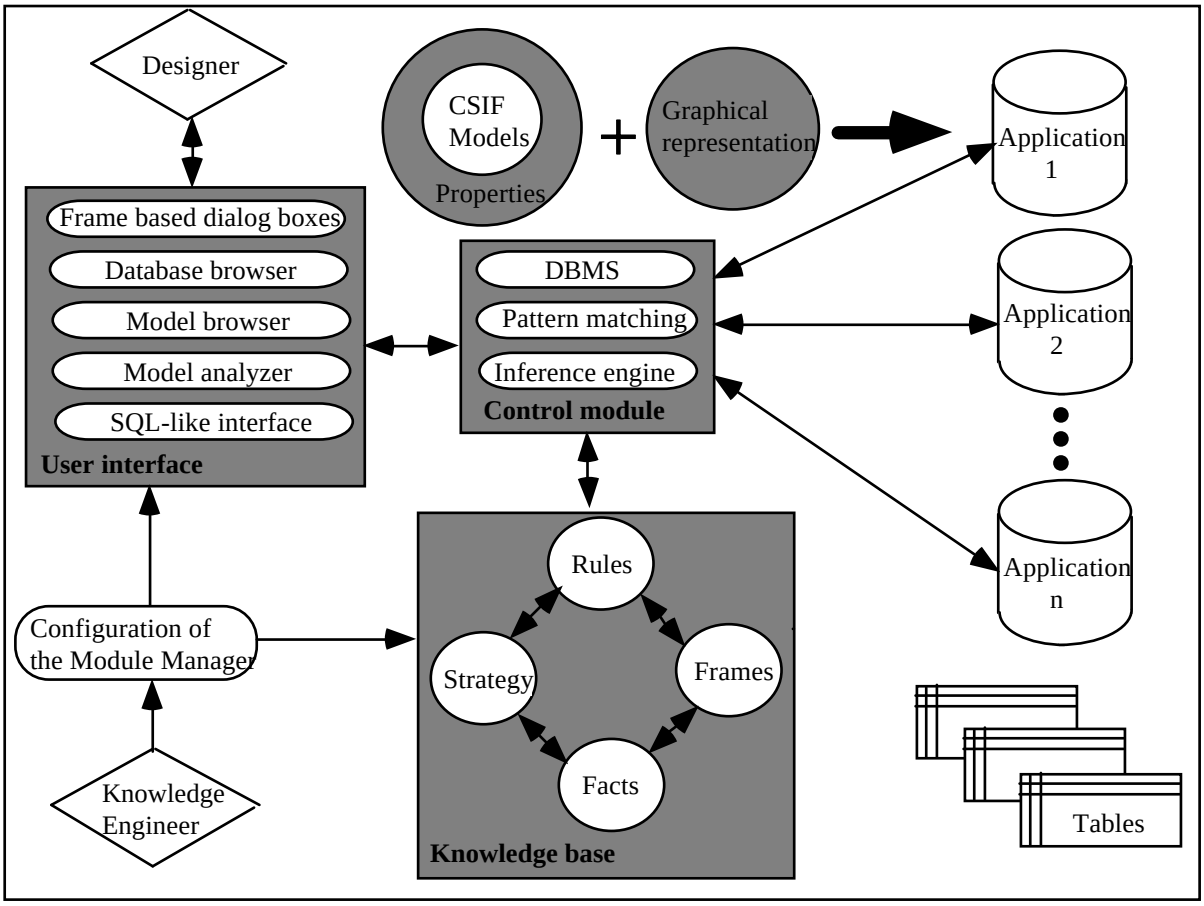


Fig. 10: The Module Manager architecture.

The Module Manager (figure 10) provides mechanisms to handle:

- a prototyping approach and expert assistance for modelling various architectures,
- a repository of models coming from high level specifications editors,
- the definition of the Knowledge Base scheme with a script based description,
- the reusability and genericity of behaviours,
- the extraction of information from the CSIF format following the classification structure,
- the reconstitution of information relevant to the chosen module in the CSIF format,
- the proposition of solutions according to the designer's requirements,
- the research of solutions through a pattern matching algorithm (see next chapter).

The control module is the heart of the tool. It communicates with the designer through a user-friendly interface, the databases and the knowledge base. The knowledge base is constituted of a repository of all the information (facts, rules, frames and strategy for rule evaluation) required to ease the design process of a hardware system. Since the Module Manager is used as a hardware specification assistant, the Inference Engine (IE) has to handle the services that are useful in helping the designer during his modelling task. The reasoning process is based on classical backward chaining (hypothesis to verify) and forward chaining (goal to achieve) inference methods. In addition to this, a pattern matching algorithm searches in the databases for the behavioural solutions that could be appropriate. This feature will be discussed in the next section. The inference engine interprets the designer's inquiries in order to suggest different possible

architectures and to propose a display of behavioural solutions that correspond to the required features.

All transactions and requests between the end-user and the Module Manager are performed using either dialogue boxes to enter the properties or a query language, such as SQL. We have also implemented a graphical browser to navigate through the structure of a model. This browser aims to give an overview of a model. When a part's behaviour is selected, the inference engine automatically invokes the corresponding editor and highlights its corresponding graphical representation. The model analyser is used to verify the conformity of the model structure with the scheme proposed by the knowledge base.

Furthermore, we give the possibility to configure the Module Manager (definition of the classification and implementation of the rules and facts) according to the customer's needs. The customer can be a designer, a lab or even a company. The Knowledge Engineer and the customer will work in close collaboration in order to define a knowledge acquisition strategy. The Knowledge Engineer will then extract, formalise and encode the knowledge in some manner, using MAGMA for the definition of the classification and Nexpert Object™ for the definition of interconnected rules. Figure 11 presents an overview of knowledge engineering. We have also provided the Module Manager with the ability to manage several independent databases. Each database is dedicated to the description of a family of models. In this way, the Module Manager can be used for the specification of many architectures from various domains.

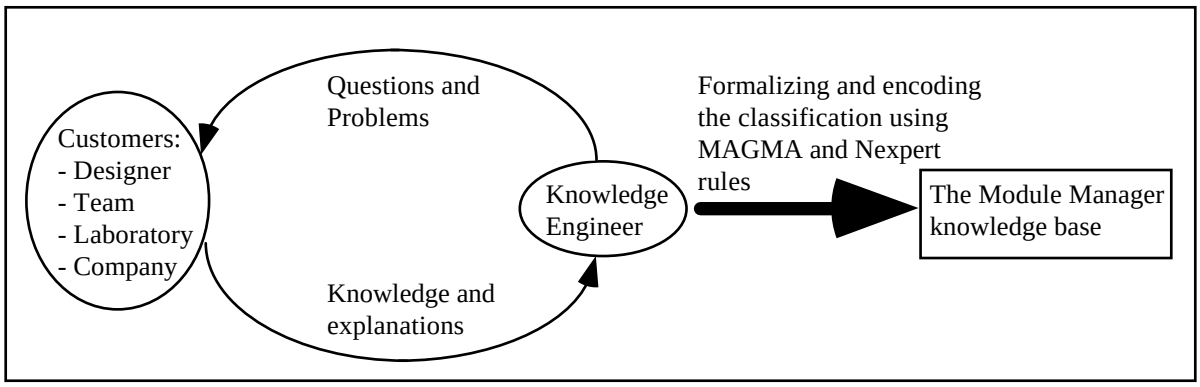


Fig. 11: Knowledge acquisition.

The Module Manager handles relational flat-file databases in which the information is stored in a table. The table consists of a set of records, each record having several fields. A record represents a logical unit of information corresponding to an instantiated model, while a field represents a property or an attribute of a model.

5 The Knowledge Representation of the Module Manager

The knowledge representation is based on two description models: a frame structure and a semantic model. Frames represent nodes of the classification and provide some verification techniques to check the consistency of the properties and some monitoring control mechanisms to supervise the storage and retrieve of models, while the semantic model infers modelling rules and decision processes.

5.1 A Frame-based system as a classification scheme

Although frames were originally proposed as a basis for understanding complex behaviours such as

visual perception or natural language. They have recently been shown to be useful for representing VLSI design [13]. The main reason for using the frame concept is to group together common knowledge about object [9]. A frame is a data structure in which properties relating to a single object, a concept, or a typical situation are grouped. The body of a frame is composed of a number of slots used to describe the properties. These properties (general information and specific properties) are the features of the behaviours belonging to a classification node. The frame also contains two instances: the CSIF behaviour and its graphical information. Each property is defined by an identifier, a domain of possible values and an optional default value. In predicate logic [16], we specify a frame as an entity-class Ec in the form:

$$Ec(x, P1, \ldots, Pn) \equiv def \ \exists x \ [P1(x) = Val1, \ P2(x) = Val2, \ldots, \ Pn(x) = Valn]$$

The properties P1 ... Pn are respectively instantiated by the values Val1 ... Valn. In this way we build a list of objects x corresponding to the various behaviours that are available for the same class. Figure 12 gives an example of a frame that gathers information about a classical memory.

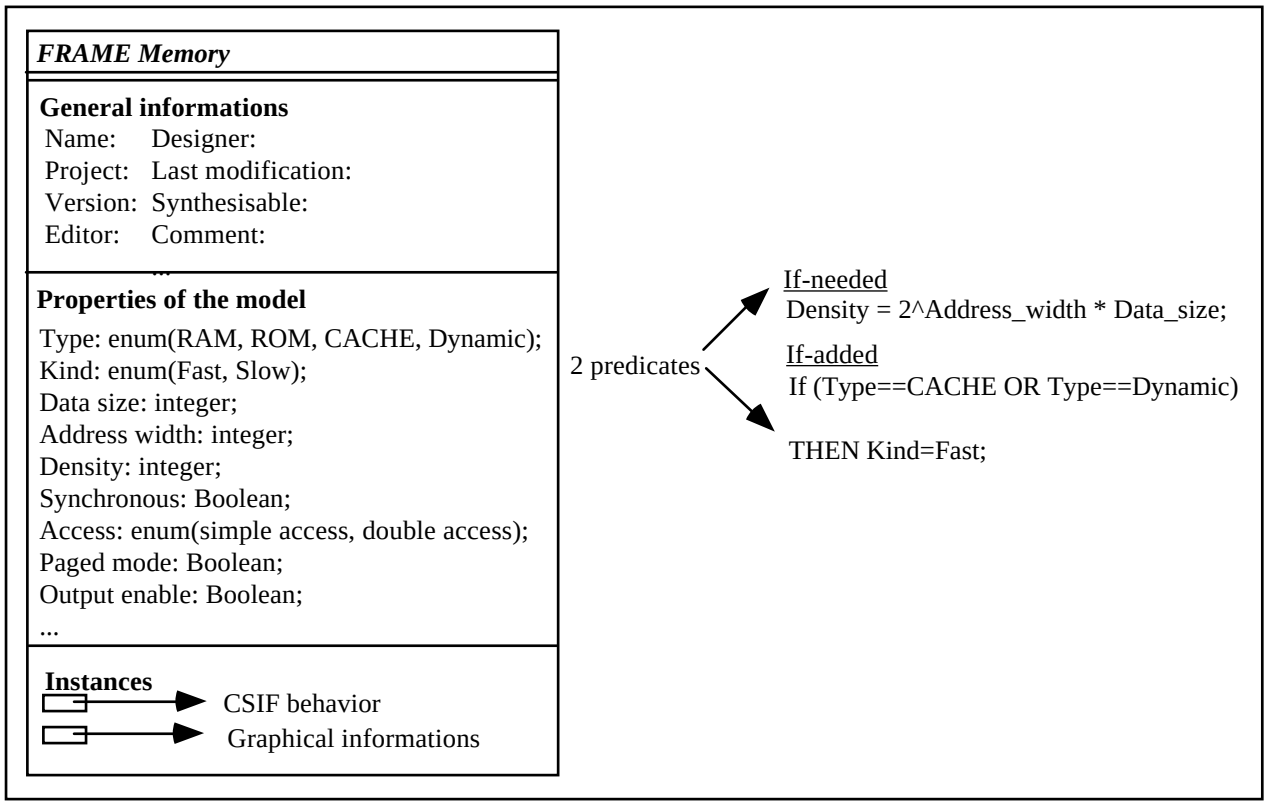


Fig.12: A frame representing a memory component.

Assigned to each slot in the Module Manager are various methods dealing with initialisation, inheritance strategy, inference strategy and consistency. In order to monitor the storage and retrieval of information in the Module Manager frame-based system, we associate with each property two optional attached predicates or demons: *If-needed* ('Order of Sources' Nexpert method) and *If-added* ('If Change' Nexpert method).

What should be done when the value of a slot or facet is required to complete an action but is not specified? Domain experts can usually list a number of potential sources from which the value can be obtained or derived. The *If-needed* predicate is used to enhance the flexibility of retrieval. For instance, when the value of a property is not directly available, we might be able to calculate its value on the basis of other information that we know about the frame. Before the value of a slot

can be read and obtained, the *If-needed* predicate must be successfully proved.

The *If-added* predicate is triggered before the value of a slot is assigned a value or changed. It is used to screen erroneous values before they are added to slots.

These two mechanisms enable the Module Manager to check whether the properties relating to different frames entered by the designer to specify or to characterise his design are consistent. These verification rules are activated whenever a behaviour is stored or retrieved in the database.

The inheritance and inference methods control the strategy and the triggering of inheritance and inference. Inheritance methods control the transfer of values declared in an object or a class to other related objects or classes. The relationship can be parent, child or objects and classes belonging to a same knowledge island. Every property of an object or class can be assigned an inference method; otherwise, it is inherited from the parent class. On the other hand, inference methods control the inference behaviour of the system; in other words, the order in which information is processed.

A frame is considered to be complete when all the slots are filled. However, when a model saved in the database is partially complete and should be achieved in multiple stages, the verification process is not broadcast in order to avoid propagation of errors through the frame system.

5.2 A semantic net as modelling guidelines

In order to help the designer with modelling guidelines rules, it turns out to be effective to use a semantic model [7] [8]. A semantic network or net is a structure for representing knowledge as a pattern of interconnected nodes and arcs. Nodes will represent classes of behaviours whereas arcs define relationships between the entities. Furthermore, we have specified six types of association (*Is-a*, *Can-be*, *May-have*, *Has-a*, *Is-linked-with*, *May-be-linked-with*) that are useful to represent all the possible configurations for modelling a family of designs into one representation. Figure 13 gives a partial example of the taxonomy for modelling microprocessor architectures.

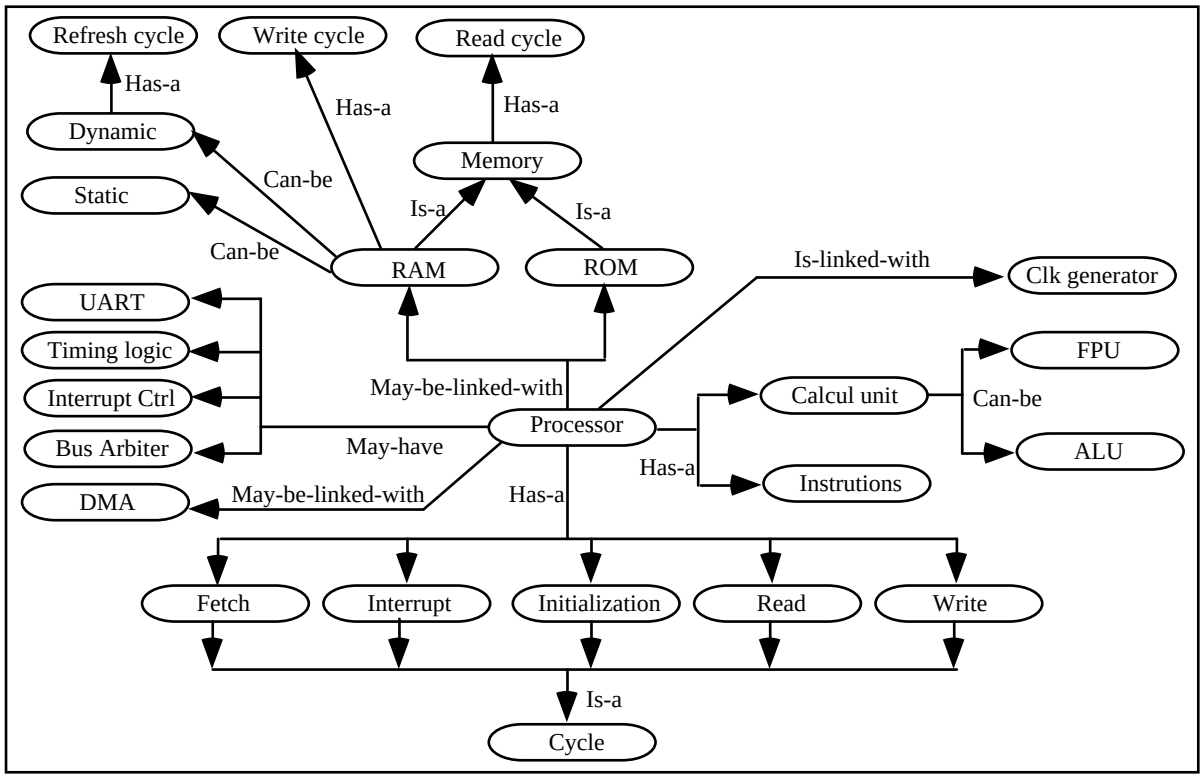


Fig. 13: An example of a semantic network (partial).

One of the problems in providing a model-theoretic account of semantic network representations is the fact that there is no uniform notation. We therefore merely illustrate the way in which one might translate the semantic network we have implemented into a proposition of first-order predicate calculus [17].

Generalisation: This association provides the concept of inheritance, a way to express constraints that define some class as a more general class of other ones. Features common to a class of objects can be grouped into a generic frame. These properties are automatically inherited by frames placed further down the classification hierarchy. We use the *Is-a* attribute to show this hierarchy:

$$Is-a(Ec(x), Ec_i(y_i) \mid i = 1,..,n) \equiv_{def} \forall x, y_i \mid i = 1,..,n \Rightarrow \exists Ec(x) \bigwedge_{i=1}^{n} \exists Ec_i(y_i)$$

$$where \quad Ec_i(y_i) \mid i = 1,..,n \subseteq Ec(x)$$

For example, *Is-A (RAM(x), Memory(y))* indicates that entities from class RAM are subset of class Memory and inherit all the properties and the concept of Memory. To simplify the notation, we will use: *Is-A(**RAM**, Memory)*.

Aggregation: Grouping classes into higher level classes is called aggregation. We define the *Has-a* statement by the following expression:

$$Has-a(Ec(x), Ec_i(y_i) \mid i = 1,..,n) \equiv_{def} \forall x \;\; Ec(x) \Rightarrow \bigwedge_{i=1}^{n} \exists y_i \;\; Ec_i(y_i)$$

For example, the fact that ALU, cycles and instructions are components of the entity-class microprocessor is represented as: *Has-a(**Microprocessor**, ALU, cycles, set_of_instructions)*.

Restriction: The restriction is a way to infer a specific choice. It also provides the concept of inheritance. The *Can-be* predicate is defined (using the XOR operator $\oplus$) as follow:

$$Can-be(Ec(x), Ec_i(y_i) \mid i = 1,..,n) \equiv_{def} \forall x \;\; Ec(x) \Rightarrow \bigoplus_{i=1}^{n} \exists! y_i \;\; Ec_i(y_i)$$

$$where \quad Ec(x) \subseteq Ec_i(y_i) \mid i = 1,..,n$$

Thus, we can say that a memory may have several architectures:

*Can-be (**Memory**, static, dynamic, read_only).*

The static, dynamic and read_only memory classes then inherit, the concept of the entity-class memory.

Possibility: The *May-have* predicate gives the possibility to select a set of entities among several classes:

$$May-have(Ec(x), Ec_i(y_i) \mid i = 1,..,n) \equiv_{def} \forall x \;\; Ec(x) \Rightarrow \bigvee_{i=1}^{n} \exists y_i \;\; Ec_i(y_i) \oplus \varnothing$$

For example, the fact that a microcontroller may contain various optional functionalities is represented by: *May-have (**Processor**, IntCtrl, Serial link, timer, bus arbiter, UART).*

Connectivity: We have also implementing two other types of association to represent interconnections between modules belonging to different frames: *May-be-linked-with* and *Is-linked-with*. A microprocessor is always connected to a clock generator: *Is-linked-with(**Processor**, Clk generator)*. The possibility to associate a DMA (Direct Memory Access) with a processor is given by *May-be-linked-with(**Processor**, DMA)*.

5.3 Reasoning process

The Module Manager acts as a case-based tool (figure 14). The more modules that are are stored within the database, the more efficient will the Module Manager be. According to the designer's requirements, the Module Manager will search in the database, using pattern matching or SQL queries, for some modules that could satisfy his (or her) needs. From there, the designer will manually modify the selected models to obtain the final design. This final design can be stored in the database for its expansion.

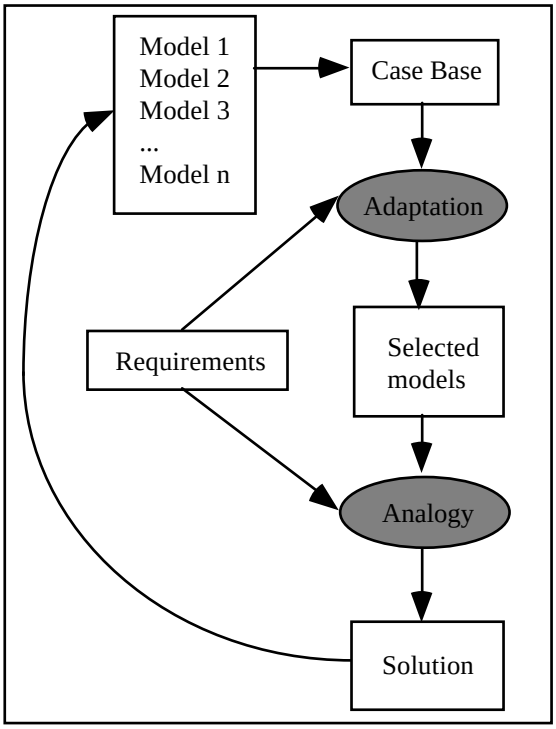


Fig. 14: A case based reasoning process.

Modelling a device with the Module Manager is carried out in an incremental fashion. The designer establishes a dialogue with the Module Manager in order to obtain the information about the functionalities and components he would like to integrate. In a first stage, the designer selects the kind of model to be specified. From this starting point, the Inference Engine (IE) will make its way through the knowledge base by interpreting the meaning of the above four types of association. From the designer's point of view, the *Is-a* association is transparent. The IE takes into account all the semantic structure placed under this link. When a *Has-a* association is encountered, the designer must characterise each type of behaviour belonging to nodes connected to this link, while the *Can-be* association forces the designer to select only one class of behaviour among several. The *May-have* association offers the possibility to include optional functionalities. Rules associated with links are used to automatically infer a decision process in order to select the right path in the semantic tree. The rules are triggered according to the values of properties belonging to the frame the Module Manager is processing. The *If-Needed* demon is used to deduce the values of

some properties. Also, in order to proceed in the evaluation of rules, the inference engine must have appropriate information on which to base its conclusion. If the values of slots incorporated in rule conditions are unknown, the system must first fetch the values to complete the evaluation by using the inheritance strategies. However, when the inference engine is not able to calculate some property values, the Module Manager will open a question window to obtain the required information from the designer. When this specification phase is completed, the IE may proceed, upon request, to a global verification of the consistency of the properties. It is also in charge of searching a set of CSIF behaviours related to the different parts of the required model.

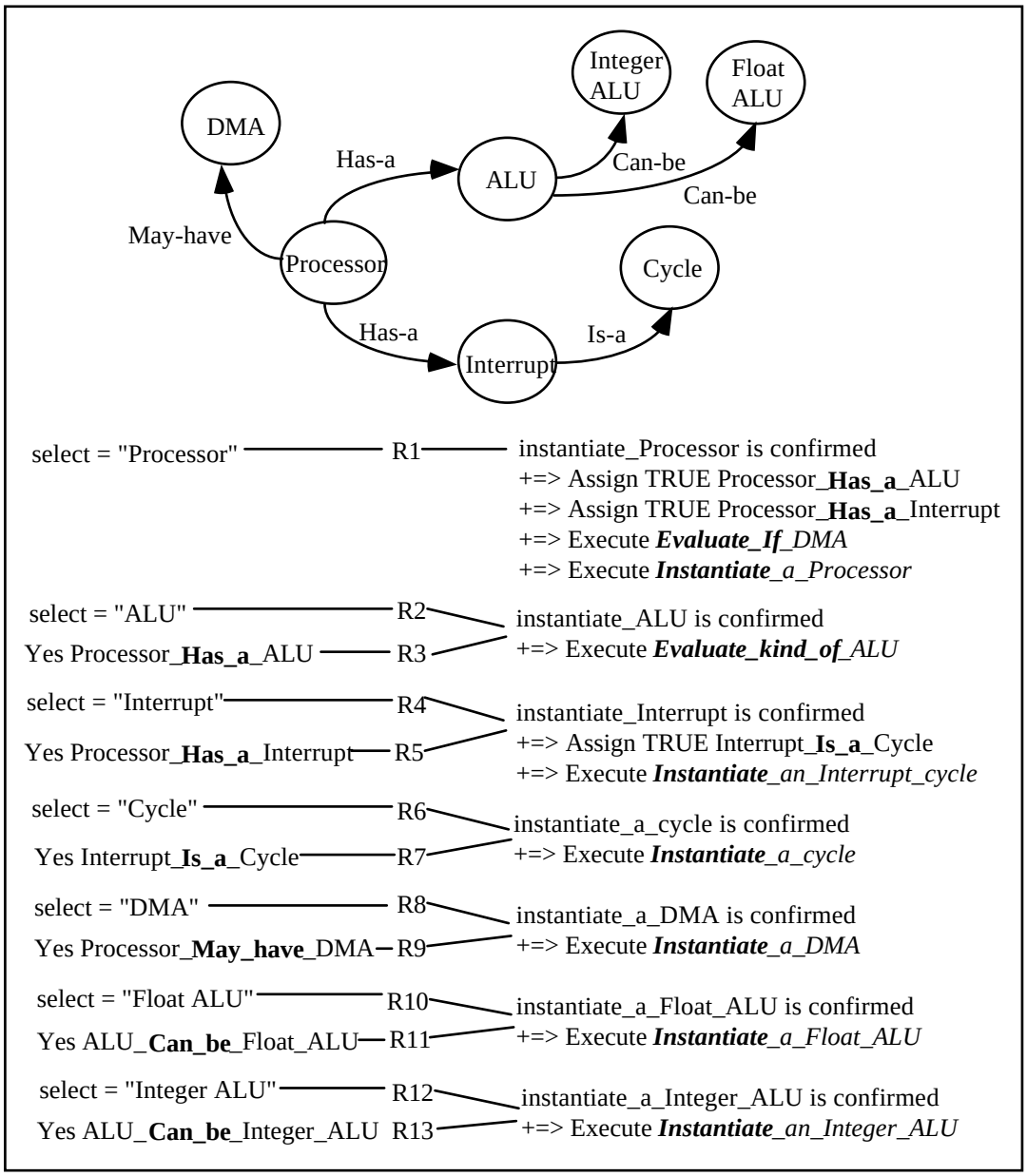


Fig. 15: Network of rules (partial).

Figure 15 gives a simple illustration of the way to translate a semantic net describing the relations between parts of a system into a network of interconnected rules. The **select** value conveys the name of the node selected by the designer and represents the starting point of the network. The **Assign** statements indicate to the inference engine the order in which the rules are to be executed. The **Execute** statements activate procedures to instantiate a model or to determine which node is to processed next. We have implemented two types of evaluate strategies. The **Evaluate_if**

procedure is applied for a *May-have* link, while the **Evaluate_kind_of** procedure acts as an XOR operator in a *Can-be* fork. In a first step, both procedures try to automatically evaluate the next rule to execute, otherwise the inference engine will refer to the designer.

5.4 Pattern matching

The pattern matching algorithm aims to take a list of instantiated models belonging to a same object and to extract from it a set of solutions that correspond to the designer's requests. In fact, this algorithm acts on the property values of the model and not on its behaviour. The algorithm is in charge of calculating the distance d(x,y) between the required properties x and the properties y of the models stored in the database. This routine is applied on the properties of type *integer*, *float*, *range*, *boolean* and *enum*. The properties of the type string are not taken into account. For the numerical values, the algorithm determines the distances d(x,y) according to the formula |x-y| / x while the distances between *enum* are calculated using priority criteria. A confidence factor (C%) given by the designer before launching the process is used to accept (d < C) or to reject the properties. These factors are attached to each property and may differ from one to another. Added to this, a tolerance T, also provided by the designer, will allow the selection of a solution according to the number of accepted conditions N verifying: N>=T.

6 Conclusion

We have described the Module Manager, an expert system able to efficiently assist the designer during a design specification phase. It is also a new approach for managing designs reusability at the system level using CSIF. The Module Manager is a flexible tool that can be used in a prototyping approach for the specification of various hardware architectures. It also reduces the time needed to generate quality models by avoiding the designer thaving to reinvent the wheel by developing new models from scratch. The advantages of using a common format are partitioning the specifications for their later reuse and providing an open CAD system to add new editors and other applications. Furthermore, all manipulation mechanisms revolve around a single representation. However, we still need to provide the Module Manager with an SQL-like interface. A future step will extend the Module Manager to a client-server architecture to distribute the database around a network of homogeneous workstations. A first example of use consists of managing a knowledge base for the specification of processor architectures.

Acknowledgements

This work is supported by the Microswiss commission for the encouragement of scientific research under grant number TR-EL-004.2.

References

[1] J.P. Calvez, Spécification et Conception des Systèmes: une Méthodologie, Editions MASSON, collection MIM 1990, 630p.

[2] Ron Waxman, "Hardware Design languages for Computer Design and Test", IEEE Computer, pp. 90 - 97, April 1986.

[3] Franz J. Rammig, "System Level Design", in Fundamentals and Standards in Hardware Description Languages, J. Mermet ed., pp. 109 - 151, Kluwer Academic Publishers, 1993.

[4] H.P. Amann, et al., "MODES: An Expert System for the Automatic Generation of Behavioural

Hardware Models", Euro VHDL'91 Proc. pp. 192 - 195, Sept.91.

[5] H.P. Amann, et al., "High-Level Specification of Behavioural Hardware Models with MODES", ISCAS'94, London, May 94.

[6] Ch. Munk, A Methodology for Designing and Using a Hardware System Specification Environment, Ph.D. thesis Nr 1309, EPFL, Lausanne, Switzerland 1994.

[7] William A. Woods, "What's in a link: Foundations for Semantic Networks", Readings in Knowledge Representation pp. 217 - 241, 1985.

[8] R. Yasdi, "Learning Classification Rules from Database in the Context of Knowledge Acquisition and Representation", IEEE Transactions on Knowledge and data engineering, Vol. 3, N0 3, pp. 293 - 306, Sept. 91.

[9] Minsky M., "A Framework for Representing Knowledge" in The Psychology of Computer Vision, Mc Graw Hill N.Y. 1978.

[10] Van den Heuvel, et al., "High-level behavioural modelling using BEMCharts: A formalisms and its application to behavioral specification of digital hardware", ECCTD'93 Proc. pp. 211 - 216.

[11] Anurag P. Gupta, et al.,"Automating the Design of Computer Systems", IEEE Transactions on CAD of integrated circuits and systems, Vol. 12, No 4, pp. 473 - 487, April 93.

[12] K.D. Mueller-Glaser, et al.," An Approach to Computer-Aided specification", IEEE journal of Solid State circuits, vol.25, No 2, pp. 335 - 345, April 90.

[13] W. Stephen Adolph, et al., "A Frame Based System for representing Knowledge About VLSI Design: A Proposal". Proc 23rd DAC'86, pp. 671 - 677.

[14] A. Hekmatpour, et al. "Hierarchical Modeling of the VLSI Design Process", IEEE Expert, pp. 56 - 70, April 91.

[15] N. Giambiasi, et al., "An Adaptative Evolutive Tool for Describing General Hierarchical Models, Based on Frames and Demons", Proc 22nd DAC'85, Los Alamitos, pp. 460 - 467.

[16] Bergmann, Moor, Nelson, The Logic Book, New York, McGraw-Hill, 1990.

[17] Nexpert Object Reference Manual, Vol. Knowledge Design, Neuron Data, Palo Alto, California, 1994.

[18] Speed Electronic, "speedCHART User's Manual", Neuchâtel Switzerland, 1994.

[19] Intel® - Peripheral Components, 1993.