An Expert Assistant for Hardware Systems Specification

Laurent Chaouat, Charles Munk, 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 as a novel approach to assist the designer in the specification of hardware systems. This flexible expert system proposes, at a high level of abstraction, behavioral solutions that match the designer's requirements. Models are selected from a repository composed of designs previously specified within the MODES environment. Thereby, the Module Manager allows their reusability and, hence their genericity. This paper focuses on the architecture and the mechanisms of the Module Manager.*

1 Introduction

The market competitiveness requires from the designer an efficient methodology [1] and a good technical background to master the increasing number and diversities of designs. Futhermore, due to the ever growing complexity of hardware systems, the designer is often confronted with a dilemma of efficiency, rapidity, quality and cost. The solution to a particular problem is far from being easy and exclusive. In order to reduce some of these difficulties, sophisticated CAD tools have significantly contributed to produce solutions according to the customer's needs. Meanwhile, generating correct solutions of good quality remains difficult. Actually, designers need a flexible tool able to propose a panorama of solutions for different domains such as microprocessors and DSP architectures, microsystems, communication protocols, multimedia, and many other systems.

The Module Manager is an expert system used as a prototyping approach to ease the design process of a hardware system. The main objective of such a tool is to reduce the time and the modeling expertise needed to generate behavioral models. It aims to assist the designer with a knowledge base to generate a set of behavioral 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 and allowing him to describe devices incrementally in a very abstract way, somewhat close to a high level datasheet description. The expert system can also be used to train the user in hardware modeling by explaining its reasoning process.

The Module Manager is involved during the specification phase of a hardware system. According to the requirements, it is able to search in a repository of previously specified models 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 adapted manually through 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 to reivent the wheel by creating new behavioral models from scratch. The Module Manager is part of the MODES (**MOD**eling **E**xpert **S**ystem) project [4] [5].

Related works

Up to now, 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 customized to represent knowledge about integrated circuits. Kinden [14] is an experimental knowledge-based intelligent environment for the VLSI design process. Micon1 [12] is a synthesis tool that aim to automate the design of computer systems.

Our research differs from these efforts by proposing an intelligent and flexible architecture able to manage the reusability of behavioral 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 implication with the Module Manager. We also give a general overview of the CSIF format, a textual representation where the specification structure of different formalisms is preserved. Since the CSIF format is an important aspect of the Module Manager, we present in section 3 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 to provide a global control over a design. Finally, we give our conclusion.

2 The Module Manager and the MODES environment.

MODES is an environment for specifying electronic devices using high level behavioral formalisms. This section presents the utility working with such an environment and the implication of the Module Manager in MODES through the CSIF format. A short overview of the CSIF format is given with 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 achieve correctly his (or her) design. An efficient design should take into account as early as possible the constraints implied by the environment into 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) modeling, iii) partitioning and iv) integration of environment constraints. The elaboration of behavioral models of hardware systems is at the heart of the system level design. Hardware description Language (HDL) have been developped to describe different views of a system, usually the behavioral and structural views, among different levels of abstraction, from the switch level to the algorithmic level [2]. However, modeling hardware systems with an HDL requires from the designer a good understanding of hardware systems and a good software programming background, especially to represent the requirements into an HDL code. Consequently, there is a need to provide a software tool which should perform the different tasks implied in the system level design.

MODES aims to provide the designer with several editors that capture specifications related to a hardware system in order to automatically generate behavioral HDL models for simulation or synthesis (Fig. 1). It also includes mechanisms that give a better interaction between the designer and the specification environment. MODES is organised around three sets of functionalities: i) the graphical capture tools for high-level specification formalisms, ii) the merge of all the specifications into a 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 informations (i.e: modeling 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.

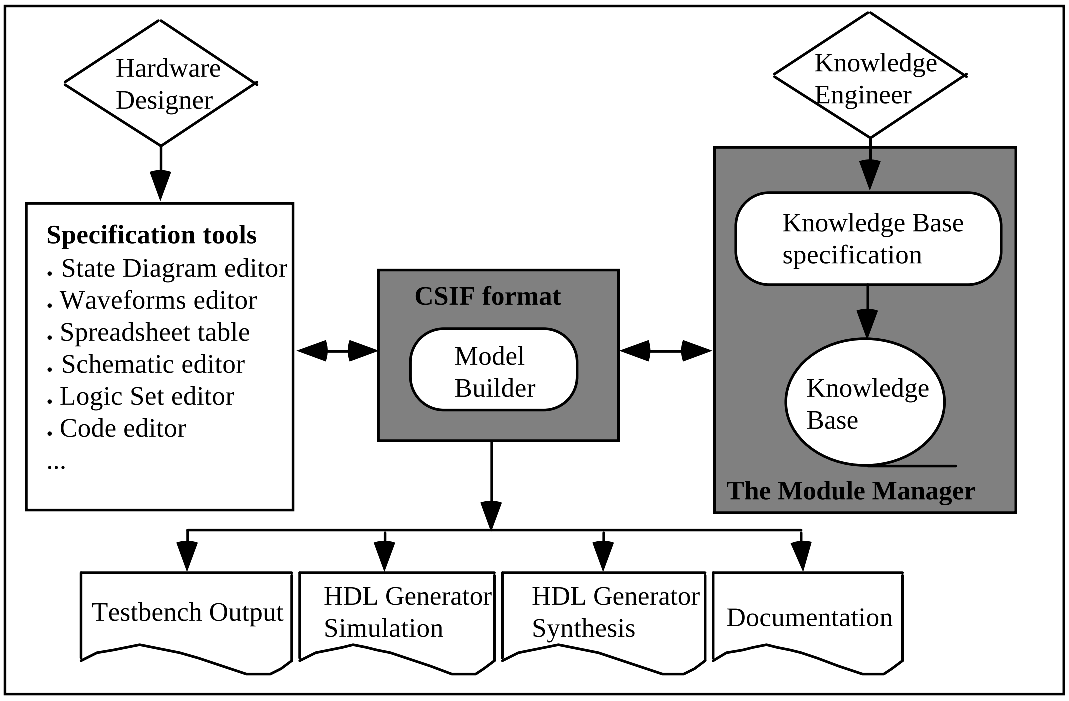


Fig. 1: The MODES Block Diagram

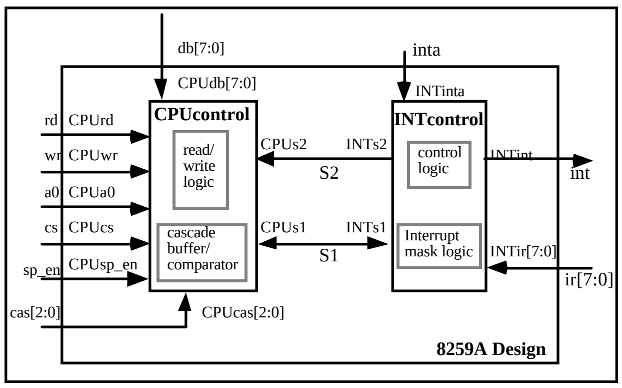
2.2 A practical solution to merge hardware system specifications: the CSIF format.

Proposing an environment with various formalisms gives the 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. It is also not easy to allow future extensions such as adding new editors or applications that may directly use the specifications to extract global properties, or to evaluate functional performances or consistency.

The CSIF format aims to go beyond these constraints. In particular, it respects the way the designer has entered the specifications (i.e: hierarchy, concurrence, partitionning, formalism ...). 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. 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, allows the genericity of a model. CSIF also offers various mechanisms to enhance its manageability.

Let's illustrate with a simple example the utility using such a format. This example taken from [6] shows how to represent in CSIF the various formalisms that we may use to specify a system. We specify the programmable interrupt controller INTEL 8259A [19] which is currently functionning with 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**). Fig. 2 shows how to easily represent the 8259A into a schematic form.

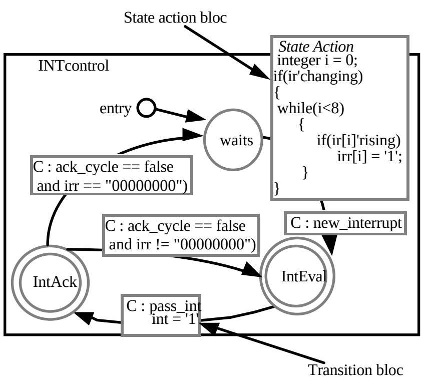
The communication between both modules, **CPUcontrol** and **INTcontrol** connected at the same level, is possible through the internal signals **s1** and **s2**. Port declarations are used to connect the corresponding component ports to the upper component interface.



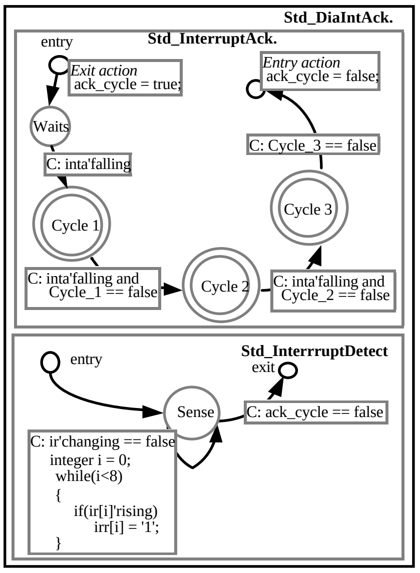
```
NetList PIC_8259A EDITOR : SCE
{
--Interface description
   logic rd     in;
   logic wr     in;
   logic a0     in;
   logic cs     in;
   logic cas[3] in;
   logic db[8]  inout;
   logic sp_en  in;
   logic ir[8]  in;
   logic int    out;
   logic inta   in;

   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. 2: The INTEL 8259A netlist description

We present in detail the description of both modules (Fig. 3, Fig. 4). The specification of the **INTcontrol** part refers to the BEMCharts formalism [10] following an extended state machine model. This module is 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. CSIF provides a control variable to re-create the control aspect of an FSM (Finite State Machine). The *StateIf* type is used to define variables that control the evolution of an FSM. The *StateIf* variable **ControlST** involved in the *Switch* statement of the **INTcontrol** module represents the control aspect of its corresponding state machine. A state of an FSM (*Case* statement) may contain four kinds of action: *Entry*, *State action*, *exit* and one or more transitions. 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 hierarchy states (**Std_InterruptDetect** and **Std_InterruptAck** which are concurrent elements). Similarly, changing the state to **"IntEval"** will disable the diagram **Std_DiaIntEval**.



a1) Top level

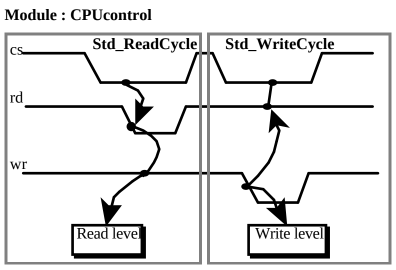


a2) Diagram DiaIntAck

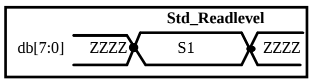
```
MODULE INTcontrol
{
-- Module Interface description
  logic  INTinta          in;
  logic  INTs2            out;
  logic  INTs1[8] inout;
  logic  INTir[8] in;
  logic  int              out;
-- Module Global Variables
  boolean        pass_in, ack_cycle;
   {   -- Module unit
     INTcontrol:(    -- 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. 3: Representation of the INTcontrol module.

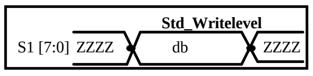
The description of the **CPUcontrol** module follows the same construction using two state machines, but comes from a waveform specification. At the top level, the **ReadLevel** and the **WriteLevel** are two diagrams giving access to a lower level of hierarchy constituted by the **Std_ReadCycle** and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



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 St_Read1 St_Write1 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. 4: Representation of the CPUcontrol module.

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 behavior to be reused. The storage of a design follows a predefined schema which specifies the different types of models and functions with their relations. This classification includes a hierarchy decomposition, properties of the behaviors, modeling and verification rules (Fig. 5). A knowledge engineer is responsible for the configuration, maintenance and upgrade of the expert system.

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

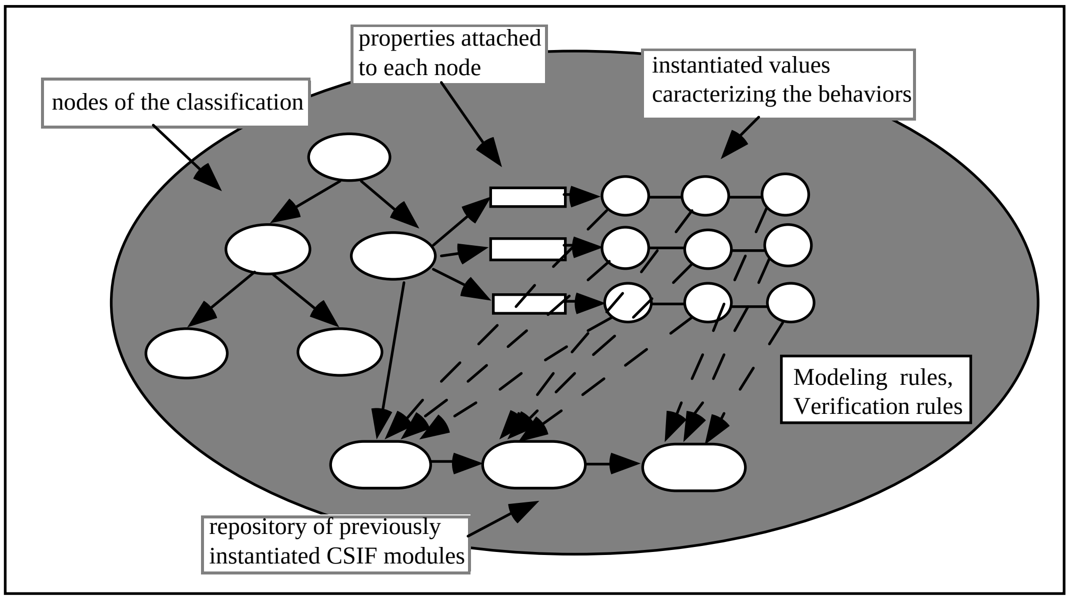

properties attached to each node
instantiated values caracterizing the behaviors
nodes of the classification
Modeling rules, Verification rules
repository of previously instantiated CSIF modules

Fig. 5: General structure of the knowledge base.

Fig. 6 gives a partial example of a kind of classification for processor architectures. The CSIF specifications are partitioned such 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 to be easily adapted for the specification of new systems. This reusability is related to several types of representation: the whole design, modules, diagrams, netlists and subprograms.

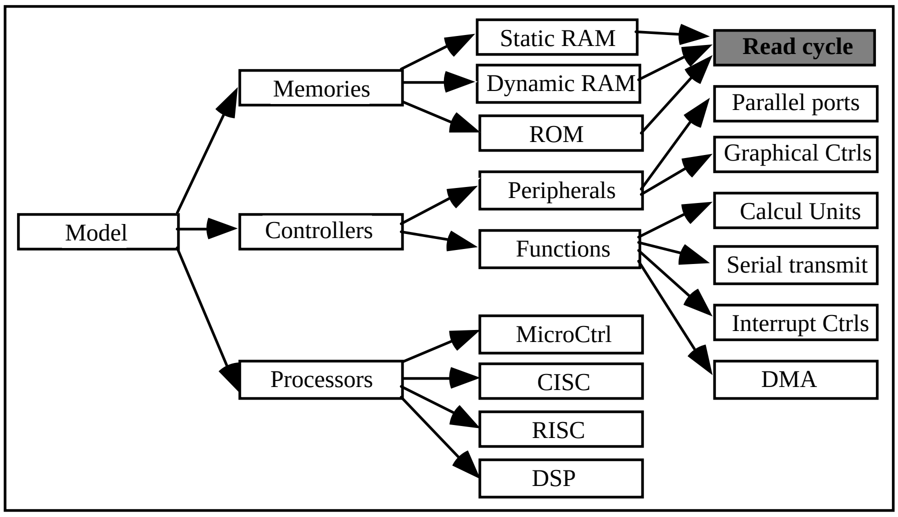


Fig. 6: An Example of classification (partial).

When all the specifications are merged into the CSIF form, a first step is to identify the behaviors that will be saved in the database. Our solution is to name with a label each of the important parts of behavior. These labels entered by the designer with the capture tools, convey the name of the behavior as well as the node of the classification tree where it has to be connected. In Fig. 7, a simple example to illustrate this mechanism is presented.

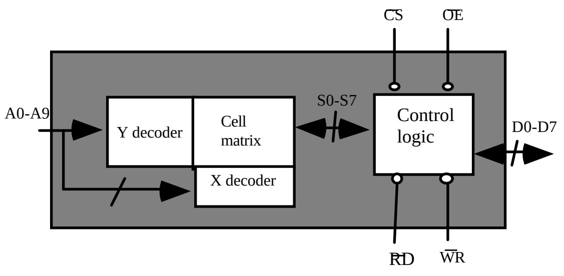
The **control logic** unit of a basic memory (a) has been specified with the waveform editor. Its CSIF representation is given in (b). This representation differs from the one we presented in Fig.4. It is essentially based on the predefined function **Check_backward()**, automatically generated by the **E**xtended **T**iming **D**iagram editor (ETD). In this *module*, we are interested to reuse the *diagram* **Read** (c). In order to identify and extract this behavior, 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 storage process of 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. With each reusable behavior, we associate the corresponding graphical representation which is also saved in the database (Fig. 8).

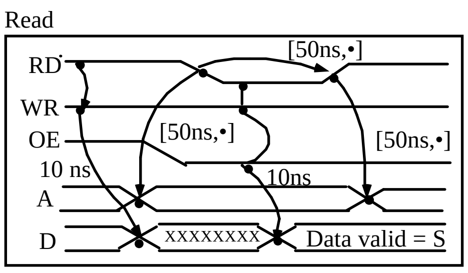
We also need to restore all the context in which the behavior is involved. To do so, a specific mechanism will dynamically search in the whole CSIF structure all the definitions of variables, signals, types, subprograms and diagrams implied in the behavior.

```
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)



(a)



(c)

Fig. 7: a) A basic RAM architecture,
b) CSIF representation of the Control logic module,
c) Specification of the Read cycle using the waveform editor.

4 The Module Manager architecture.

The Module Manager is a mixed software architecture based on a Database Management System (DBMS) written in C++ and on an expert module (knowledge base and inference engine) able to propose 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 informations required to facilitate the design process of a system. In this context, it also manages the different versions of a model. Modeling mechanisms offer the necessary informations to create new behavioral models by reusing previously instantiated designs.

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

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

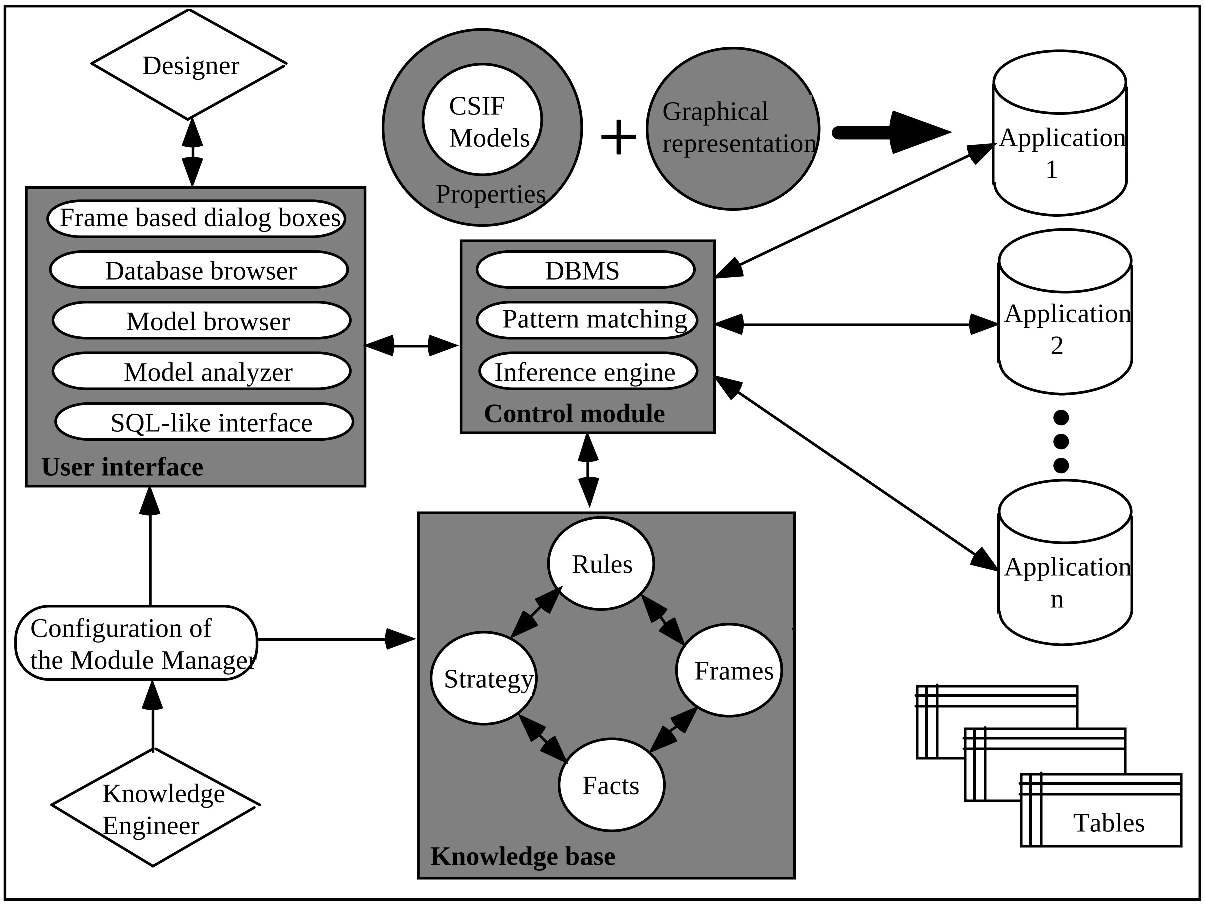


Fig. 8: The Module Manager architecture.

The Module Manager (Fig. 8) provides mechanisms to handle:

- the definition of the Knowledge Base schema,

- the reusability and generecity of behaviors,

- the extraction of information from the CSIF format following the classification structure,

- the reconstitution of informations relative to the chosen module in the CSIF format,

- the research of solutions.

The control module is the heart of the tool. It communicates with the designer through a friendly user-interface, the databases and the knowledge base. The knowledge base is constituted of a repository of all the informations (facts, rules, frames and strategy for rules 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 useful to help the designer during his modeling task. The reasoning process is based on classical backward chaining (hypothesis to verify) and forward chaining (goal to achieve) inference methods. This feature will be moved on 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 behavioral solutions that correspond to the required features.

All transactions and requests between the end-user and the Module Manager are performed using dialog 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 behavior part is selected, the inference engine automatically invokes the corresponding editor and highlights its corresponding graphical representation. The model analyzer is used to verify the conformity of the model structure with the schema 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 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, formalize and encode the knowledge, using MAGMA for the definition of the classification and Nexpert Object™ for the definition of interconnected rules. Fig. 9 presents an overview of knowledge engineering. We have also provided the Module Manager with the ability to manage several independant 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.

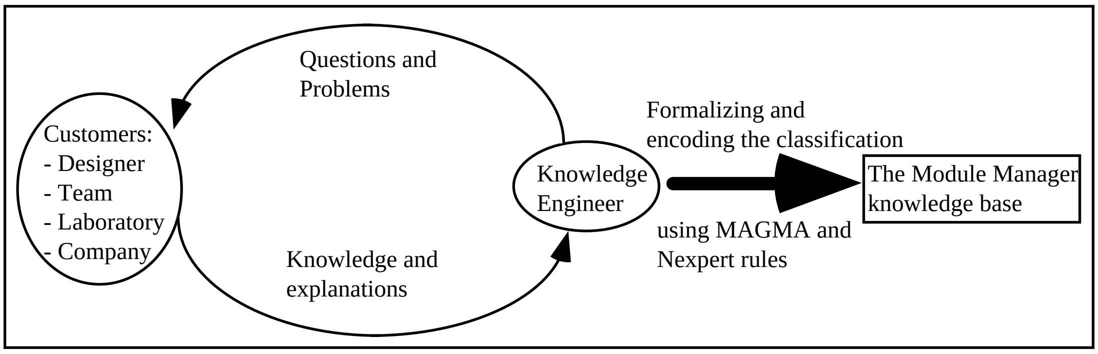

Questions and Problems
Customers:
- Designer
- Team
- Laboratory
- Company
Knowledge and explanations
Knowledge Engineer
Formalizing and encoding the classification
using MAGMA and Nexpert rules
The Module Manager knowledge base

Fig. 9: Knowledge acquisition.

The Module Manager handles relational flat-file databases where the informations are stored in a table. The table consists of a set of records with 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 modeling rules and decision processes.

5.1 A Frame-based system as a classification schema.

Frames were originally proposed as a basis for understanding complex behaviors such as visual perception or natural language. Recently, they have been shown to be a useful mean for representing VLSI design [13]. The main reason to use 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 are the features of the behaviors belonging to a classification node. 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 respectivally instantiated by the values Val1 ... Valn. In this way, we build a list of objects x corresponding to the various behaviors that are available for a same class.

Assigned to each slot in the Module Manager are various methods dealing with initialization, inheritance strategy, inference strategy and consistency. In order to monitor the storage and retrieval of informations in the Module Manager frame-based system, we associate with each property two optional attached predicates or deamons: *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 kinds of 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 proven.

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.

With these two mechanisms, the Module Manager is able to check if the properties relating to different frames entered by the designer to specify or to characterize his design are consistent. These verification rules are activated whenever a behavior is stored in the database or retrieved.

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 assign an inference method; otherwise, it is inherited from the parent class. On the other hand, inference methods control the inference behavior 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 broadcasted to avoid propagation of errors through the frame system.

5.2 A semantic net as modeling guidelines.

In order to help the designer with modeling 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 behaviors whereas arcs define relationships between the entities. Also, we have specified four types of association (*Is-a*, *Can-be*, *May-have*, *Has-a*) useful to represent all the possible configurations for modeling a family of designs. Fig. 10 gives a partial example of taxonomy for modeling microprocessor architectures.

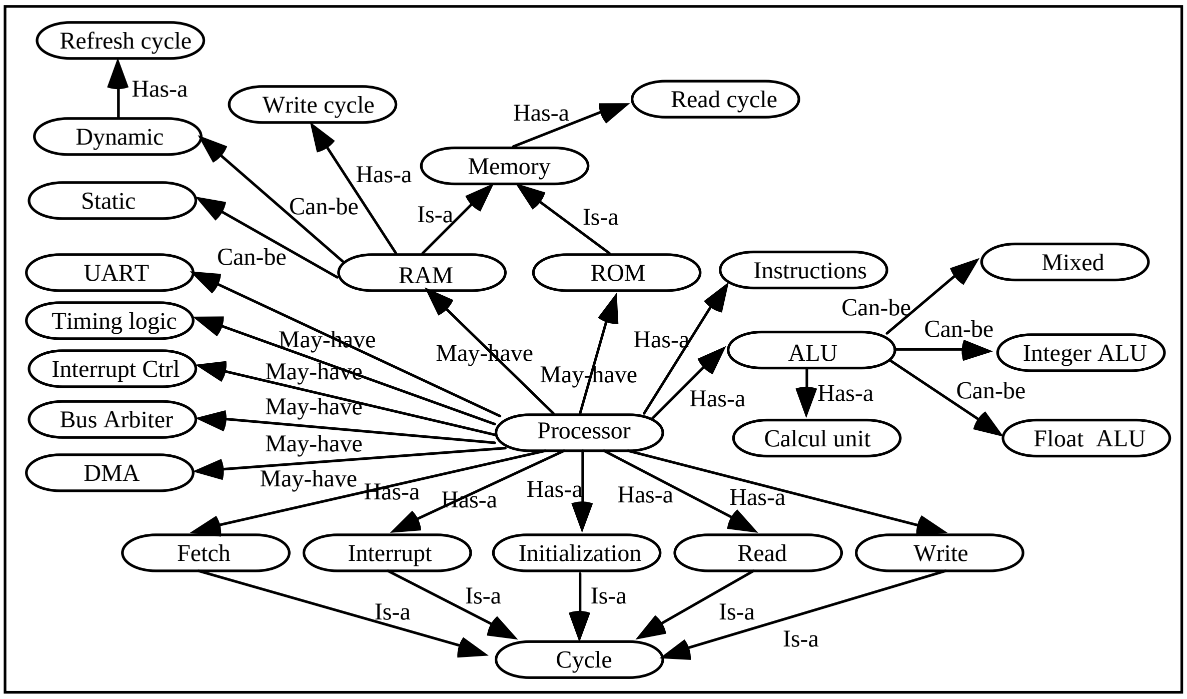


Fig. 10: An example of semantic network (partial).

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 [16].

Generalization: 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:

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

$$where \;\; Ec_i(y_i) \,|\, \mathrm{i}=1,..,\mathrm{n} \subseteq Ec(x)$$

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:

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

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:

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

$$where \;\; Ec(x) \subseteq Ec_i(y_i) \,|\, \mathrm{i}=1,..,\mathrm{n}$$

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:

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

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

5.3 Reasoning process.

Modeling a device with the Module Manager is carried out in an incremental fashion. The designer establishes a dialog with the Module Manager in order to get the informations 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 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 characterize each type of behavior belonging to the nodes connected to this link, while the *Can-be* association forces the designer to select only one class of behavior 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 way 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* deamon is used to deduce the values of some properties. Also, in order to proceed in the evaluation of rules, the inference engine must have the appropriate information on which to base its conclusion. If the values of slots incorporated in rule conditions is 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 informations from the designer. When this specification phase is completed, the IE may proceed upon request to a global consistency verification of the properties. It is also in charge of searching a set of CSIF behaviors related to the different parts of the required model.

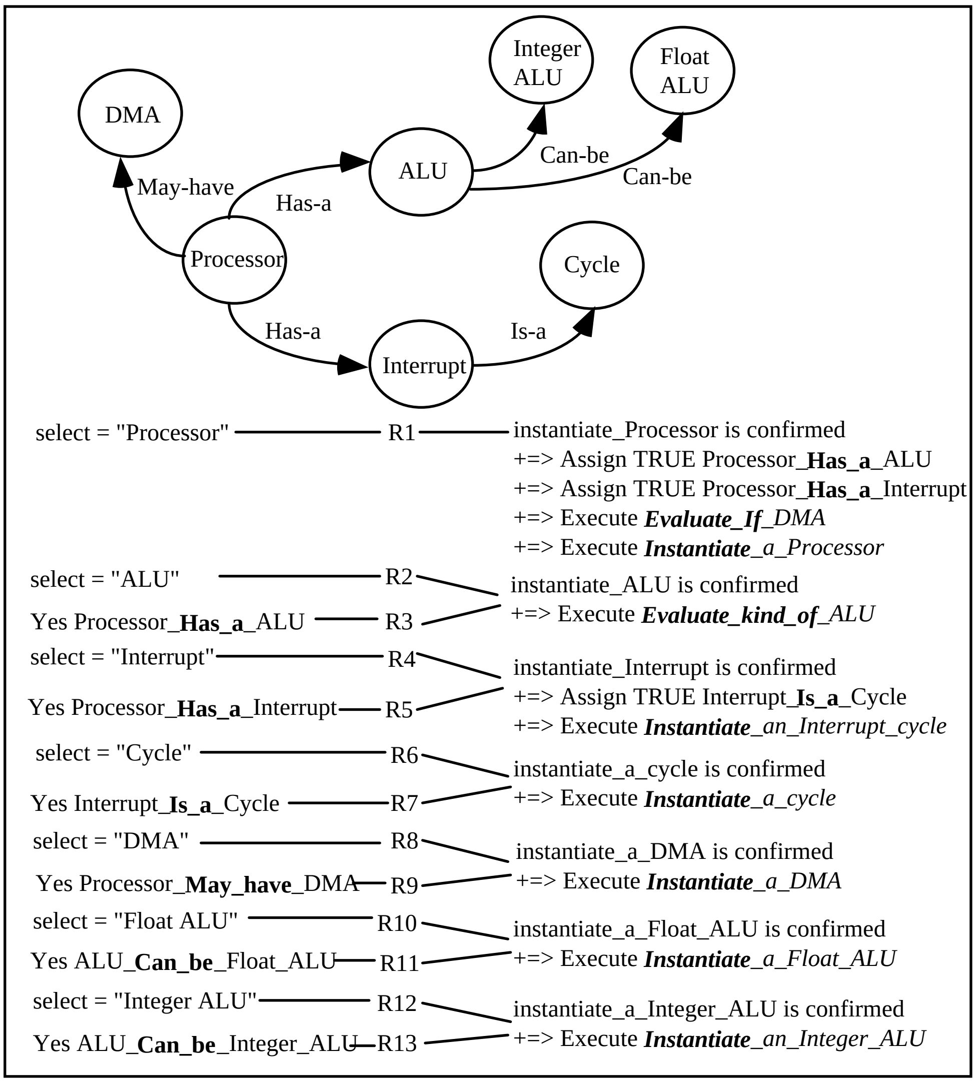

DMA
Integer ALU
Float ALU
ALU
May-have
Has-a
Can-be
Can-be
Processor
Cycle
Has-a
Is-a
Interrupt
select = "Processor"
R1
instantiate_Processor is confirmed
+=> Assign TRUE Processor_**Has_a**_ALU
+=> Assign TRUE Processor_**Has_a**_Interrupt
+=> Execute ***Evaluate_If****_DMA*
+=> Execute ***Instantiate****_a_Processor*
select = "ALU"
R2
Yes Processor_**Has_a**_ALU
R3
instantiate_ALU is confirmed
+=> Execute ***Evaluate_kind_of****_ALU*
select = "Interrupt"
R4
Yes Processor_**Has_a**_Interrupt
R5
instantiate_Interrupt is confirmed
+=> Assign TRUE Interrupt_**Is_a**_Cycle
+=> Execute ***Instantiate****_an_Interrupt_cycle*
select = "Cycle"
R6
Yes Interrupt_**Is_a**_Cycle
R7
instantiate_a_cycle is confirmed
+=> Execute ***Instantiate****_a_cycle*
select = "DMA"
R8
Yes Processor_**May_have**_DMA
R9
instantiate_a_DMA is confirmed
+=> Execute ***Instantiate****_a_DMA*
select = "Float ALU"
R10
Yes ALU_**Can_be**_Float_ALU
R11
instantiate_a_Float_ALU is confirmed
+=> Execute ***Instantiate****_a_Float_ALU*
select = "Integer ALU"
R12
Yes ALU_**Can_be**_Integer_ALU
R13
instantiate_a_Integer_ALU is confirmed
+=> Execute ***Instantiate****_an_Integer_ALU*

Fig. 11: Network of rules (partial).

In Fig. 11, we simply illustrate 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 the inference engine the order of rules execution. The **Execute** statements activate procedures to instantiate a model or to determine which node is next to process. 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.

6 Conclusion.

We have described the Module Manager as an expert system able to efficiently assist the designer during a design specification phase. It is as well a new approach for managing designs reusability at the system level. The Module Manager is a flexible tool that can be used for the specification of various hardware architectures. The advantages 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. Also, 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 microprocessor architectures.

Aknowledgements

This work is supported by the Microswiss commission of scientific research encouragement under grant number Nr 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 Standarts 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 formalism 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, N0 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, n0 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.