Three Decades of Hardware Description Languages in Europe

Jean Mermet, Peter Marwedel, Franz J. Rammig, Cleland Newton, Domminique Borrione, Claude Lefaou

Reprinted from
JOURNAL OF ELECTRICAL ENGINEERING AND INFORMATION SCIENCE
Vol. 3. No. 6, December 1998

Three Decades of Hardware Description Languages in Europe

Jean Mermet, Peter Marwedel, Franz J. Ramming, Cleland Newton, Domminique Borrione, and Claude Lefaou

Abstract

This paper binds together a collection of short presentations on Hardware Description Languages(HDLs) developed in Europe and provides a view of the history of HDLs during the last three decades. This historical review wants to present the ideas, conceived in these previous languages, which are now implemented in the standard languages. Furthermore, this paper will highlight those early concepts which yet need to be implemented in the evolving standards or could provide a way to unify them (like VHDL or Verilog or SDL) within a formally defined multi-language environment. Among a large number of European works over 3 decades, we have selected a sample from different countries France, Germany, U.K, Italy, which have been implemented and used reliably in various segments of the industry. The selected HDLs, with the date of origination, are: CASSANDRE (1967), MIMOLA (1977), DACAPO (1977), ELLA(1979), ART (1980), and CASCADE (1981). We do not pretend to any exhaustive review, which is not the goal of this presentation, and have consciously left aside several works as valuable as those selected. We have not addressed for example « synchronous languages » very well developed in France, such as ESTEREL, LUSTRE or SIGNAL. Several other works existed in Germany, such as KARL, which was popular in the eighties, and benefits from a large bibliography or REGLAN. We should mention also among those HDLs not presented here CONLAN (a major international standardization effort involving a notable European contribution). We have tried to compare the main features of the chosen languages according to a list of criteria and briefly identify those which are still missing in the recognized worldwide standards.

Keywords : Hardware Description Language, CASSANDRE, MIMOLA, DACAPO, ELLA, ART, CASCADE

I. Introduction

This collection of short presentations on HDL's developed in Europe provides a view of the history of Hardware Description Languages (HDL) during the last three decades. The presentations are aimed at complementing the two paper series published in the Computer Society "Design & Test" journal in 1992 That series presented mainly works done in the USA together with a survey of VHDL related topics.

This historical review wants to present the ideas, conceived in these previous languages, which are now implemented in the standard languages. The review may explain partially why VHDL has become the universally adopted standard in Europe faster than in the USA. Furthermore, this paper will highlight those early concepts which yet need to be implemented in the evolving standards or could provide a way to unify them (like VHDL or Verilog or SDL) within a formally defined multi-language environment.

Among a large number of European works over 3 decades, we have selected a sample from different countries which have been implemented and used reliably in various segments of the industry. The selected HDLs, with the date of origination, are: CASSANDRE (1967), MIMOLA (1977), DACAPO (1977), ELLA(1979), ART (1980), and CASCADE (1981).

We do not pretend to any exhaustive review, which is not the goal of this presentation, and have consciously left aside several works as valuable as those selected. We have not addressed for example « synchronous languages » very well developed in France, such as ESTEREL, LUSTRE or SIGNAL. Several other works existed in Germany, such as KARL(G12), which was popular in the eighties, and benefits from a large bibliography. We should mention also among those HDLs not presented here, REGLAN (G20) (major contributor was Piloty) and CONLAN (several authors, but notable in Europe are Piloty and Borrione).

CONLAN was an international effort of 6 scientists from

Jean Mermet is with TIMA-UJF, Bat.C, 120 rurede la piscine, BP53, 38041 Cedex 9, Grenoble, France.

Europe and USA that took place between 1975 and 1981 and has been reported in the D&T series mentioned above. CONLAN had a significant impact on further language developments, in particular the early specification of VHDL.

II. CASSANDRE : Computer Aided Simulation, Synthesis, ANalysis, Description and REalisation of digital systems.

Contributed by Jean Mermet, TIMA Lab., Grenoble, France

1. Introduction

CASSANDRE was designed between July 1966 and March 1968 [2.1], in cooperation with F. Lustman [2.2] a former university colleague who had joined the Thomson company. The papers by Y. Chu at University of Maryland [2.5], Schlaeppi [2.6] and K. Iverson at IBM (Yorktown Heights) [2.7], and the discussions I could have with them in 1968, were very helpful to validate the main choices of CASSANDRE.

The definition of Cassandre was driven by the following requirements:

- An easy to learn programming language syntax associated with precise hardware semantics.
- Modularity with natural and powerful entities to describe hierarchical structures
- Primitives to describe the behavior of circuits at several levels of abstraction: from architectural level down to Boolean equations.
- Automatic compilation into a simulation model and also into a directly synthesizable logic network.
- Finite State Machine as a primitive, allowing any description to be a hierarchy of automata.
- Abilities to describe any format of micro-instructions.
- Mechanisms to allow a list of micro-instructions described in CASSANDRE to be automatically encoded in the ROM of a microprocessor architecture also described in CASSANDRE.
- Data-flow concurrent statements
- Variables (and operators) with an arbitrary number of dimensions (vectors, arrays, cubes, ...).

2. Implementation

During years 1969 and 1970, a first implementation of CASSANDRE was realized in assembly language on an IBM 360/ 67 machine. A simulation system was developed in 2 versions: synchronous and asynchronous.

In the synchronous version one or several clocks were used (declared clock) to trigger memory elements loading. In the asynchronous simulator (more accurate but an order of magnitude slower) the loading could be triggered by signal edges through a derivation operator and new clocks could be derived from the existing ones through delays. A RT level synthesis was also developed later, in 1974.

Several industrial partners started to use CASSANDRE, essentially for the design of new computer architectures and micro-programs (Philips, Ordoprocesseur, Crouzet, Institut Français du Pétrole, ...). These collaborations made it possible to improve the implementation and to obtain a robust system. The marketing rights were given to a software company, but during the seventies, the market did not exist for RT languages such as CASSANDRE, which, consequently, never really became a commercial product.

3. Basic features of CASSANDRE

• Notations

The syntax of CASSANDRE is Algol-like, with if conditions, go to for state transitions, for loops to describe repetitive structures, Algol identifiers and usual boolean and arithmetic operators working on scalar and array type variables.

• Modules

The most important notion is the unit, which represents a hardware module (a precursor to the VHDL entity). To my knowledge, CASSANDRE was the first entirely modular HDL ever implemented.

Units instantiated inside other units as components of the architecture of these units, were declared external (as externally described). In CASSANDRE, as in VHDL, it was possible to mix functional and structural constructs in the same description. A CASSANDRE unit instantiation was an arbitrary hierarchy of nested units. But each of these units could be used separately as an autonomous model and elaborated for independent simulation or synthesis.

Example 2.1:
(↑ shiftoperator, V/ Iverson's reduction with "or", & concatenation)

```
Unit ADDER (H,M(1 : n) ; R (1 : n+1)) ;        (n is generic)

Register D, OF, L, ADD, N(1 : n), A(1 : n) ;

STATE1 : <H> N<--M, D<--0, OF<--0,          goto STATE2 ;

STATE2 : <H> A<--N # A, OF&N<--1 ↑ N^A,
             D<--V/ N^A,                    goto STATE3;

STATE3 : <H> if D=0 then (L<--OF, ADD<--0, goto STATE4)
                  else (D<--0,             goto STATE2);

STATE4 : <H> if ADD=0 then                  goto STATE4
                  else (L<--0,              goto STATE1);
```

• Automata

To describe the sequential behavior, the notion of automaton was taken as a primitive in CASSANDRE. The statements go to and state allowed a user to load or read an implicit

state register. Symbolic state names were used to label the instructions controlled by this state (see Example 2.1 and Fig. 2.1). The do instruction provided a mechanism to externally force the automaton in any state, making it possible to build up a hierarchy of automata superposed on top of the hierarchy of units.

• Data-flow statements

In CASSANDRE, all statements are concurrent. Sequencing must therefore be described explicitly using FSMs, (but, in the asynchronous version, transport delays could impose some order to signal assignments). In Example 2.1 called ADDER, A, OF, N and D are assigned concurrently in STATE2.

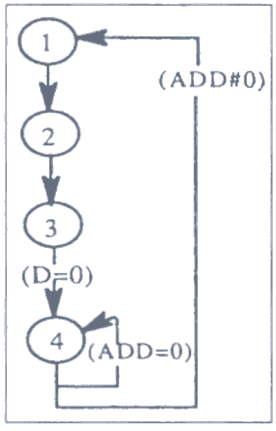

Fig. 2.1. State Chart corresonding to Example 2.

• Variables and operations

There are 3 types of variables: Register (or memory), Signal and Pulse. A pulse, used to generate clocks, can be derived from a rising or falling edge of a signal. Variables and operators are arrays with an arbitrary number of dimensions. The basic operations are the 2 operands boolean functions. They apply, as an array of operators, to each couple of corresponding elements of 2 compatible variables (dimensional compatibility is checked during compilation). There are also monadic wire-crossing operations (like shift or rotate) which apply to all vectors of the first dimension of a variable. Because the transposition operator () can move any dimension of an array to the first position , these operations are fully general (see Example 2.2).

Example 2.2:

Following the declaration:	Signal: S(1:4,1:5), Z(1:4);
Z(1) := ^/S *means:*	Z(1) := S(1,1)^S(1,2) ^S(1,3)^S(1,4)^S(1,5)
	.....
	Z(4) := S(4,1)^S(4,2)^S(4,3)^S(4,4)^S(4,5)

• Generics

Regular structures of registers, signals or units can be generated by the "faire" statement ("faire" means "generate" which is used in Example 2.3 and Fig. 2.2 for easier understanding). If statements can be nested with generates to create more sophisticated constructs.

Example 2.3

```
External : U1(1,1;1,1), U2(1,1;1,1);
(2 components with 2 inputs and 2 outputs each)

Signal : A(1:9,1:2,1:9);
(3 dimensionnal signal array)

for K equal 1 step 1 to 8 generate

for J equal 1 step 1 to 8 generate

(if (2 Rem ( (K-1) x 8 + J)=0)
then
U1 (A (K, 1, J), A (K, 2, J) ; A (K+1, 1, J), A(K, 2, J+1) )
else
U2 (A (K, 1, J), A (K, 2, J) ; A (K+1, 1, J), A (K, 2, J+1)))
```

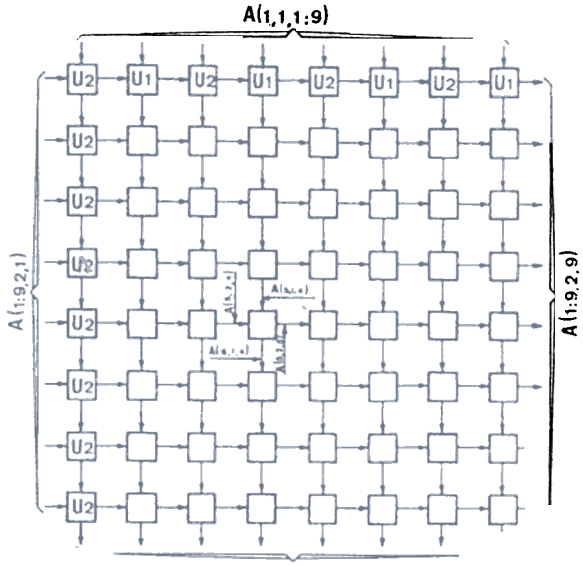

Fig. 2.2. Matrix of cells

4. The simulation algorithm

The CASSANDRE RT level simulator was not event driven but **cycle based**, and thus quite fast. The description was compiled into simulation structures generated in the 360 assembly code. An elegant stabilization algorithm, conceived by F. Lustman, was applied to the whole list of concurrent statements contained in the CASSANDRE simulation model. To avoid scanning so many instructions where nothing happens, flags were positioned which made it possible to skip large pieces of latent parts of the model.

The average number of stabilization cycles (around 2.5) happened to be surprisingly close to the optimum (2) which corresponds to a complete causal order of all the statements.

This average figure was derived from « reasonable » models (it is of course always possible, for example by moving the carry of an adder in the wrong direction, to make stabilization longer).

Then the economy of event list structures and file management, and the speed of the compiled code, made this mechanism usually much faster than event driven solutions used in almost all logic simulators. The algorithm was completed by a very simple criteria aimed at detecting permanent oscillations.

A CASSANDRE « asynchronous » simulator was also developed to accommodate models using transport delays. This simulator was working on a simulation structure close (although simpler) to the VHDL one.

Queues were associated to the drivers of delayed signals. The scheduling principle was « event-driven » and this version of the simulator was about 10 times slower than the « synchronous » version.

5. Limitations of CASSANDRE and further extensions

The main limitations which appeared in the use of CASSANDRE were:

- The lack of arithmetic operations in the behavioral description part. Arithmetic operations existed but they were only associated to generics (behavioral arithmetic would be introduced later with the LASCAR [6.1] language).
- Sequential algorithmic descriptions were not allowed. The sequencing had to be explicit, through the use of (synchronous) FSM or asynchronous delays. This was not satisfactory at system level and became later the motivation to introduce the LASSO [6.2] language. LASSO was going to offer a more general explicit description of control through the use of generalized Petri nets.
- Zero delay loops were not statically detected. These loops were in fact sometimes real sequential circuits but more often design mistakes. The later case could be detected by simulation oscillations, but if the simulation benchmarks never caused these oscillations to occur, then the mistake could remain hidden. It was a feature of the CASCADE [G.5] system, later, to perform a static analysis of loops at compilation time, using the TRAJAN algorithm on a data-flow graph.
- CASSANDRE was not portable, due to its implementation in 360 assembly language.

CASSANDRE References

[1.1] J. MERMET, Le langage CASSANDRE, Rapport final (225 pages). Contrat DGRST 660069, mars 1968.

[1.2] F. LUSTMAN and J. MERMET, CASSANDRE un langage de description de machines digitales, Revue Bleue de lAFIRO, N°15, 1969.

[1.3] F. ANCEAU, P. LIDDELL, J. MERMET, and C. PAYAN A language to describe digital systems, applications to logic design,3rd Symposium on Computer and Information Sciences, Miami, pp 179-204, 18-20 December 1969.

[1.4] G. BOGO, A. GUYOT, A. LUX, J. MERMET, and C. PAYAN, CASSANDRE and the computer aided logical systems design, IFIP World congress, august 23-28, 1971, Lubljana.

[1.5] Y. CHU, An algol-like Computer Design Language, Comm. of ACM, october1965, pp 607-615.

[1.6] H.P. SCHLAEPPI, A formal language describing machine logic, timing and sequencing (LOTIS), IEEE trans. on electronic computers, vol EC-13 pp 439-448, aug. 1964.

[1.7] A.D. FALKOFF, K.E. IVERSON, and E.H. SUSSENGUTH, Formal description of system 360, IBM sys. J. vol. 3, pp 198-262, 1964.

[1.8] Y. BRESSY, B.T. DAVID, Y. FANTINO, and J. MERMET, A Hardware compiler for interactive realization of logical systems described in CASSANDRE, International Symposium on computer hardware description languages and their applications, New-York, September 1975.

III. MIMOLA: An HDL Designed for Synthesis

Contributed by P. Marwedel, University of Dortmund, Germany

MIMOLA is a hardware design language which was defined especially for synthesis, including architectural synthesis, machine code synthesis (compilation), and test synthesis. Due to its origin, the full language is synthesizable (and also simulatable). Special language elements have been incorporated into the language to support synthesis. As a result, the language provides a homogenous environment for synthesis tools.

1. Origin of the MIMOLA language

Back in 1976 and independently of others following similar lines of thought, G. Zimmermann proposed a new design technique, which is now called *high-level synthesis* or *architectural synthesis*. To support this new design technique, he defined the first version of a new hardware design language called MIMOLA [3.15]. MIMOLA stands for 'machine independent micro-programming language. MIMOLA was designed as an input language for synthesis. Hence, it does not describe semantics in a way which would make sense only during simulations. In the years that followed, the syntax of MIMOLA was very much influenced

by other computer languages such as PASCAL and VHDL. With respect to semantics and description capabilities, the language which comes closest is Hardware-C [3.4]. A major set of innovative tools were build around the language. The first set of tools, called MSS1 (MIMOLA Software System 1) was used by Honeywell for application studies in the early 80s. The design of a second set of tools, called MSS2, was started during the same period. MSS2 has been used by some academic design groups until the mid-90s. Important milestones of the MIMOLA project are listed in Table 3.1. Table 3.

Table 3.1 Evolution of the MIMOLA language and related tools

1977	G. Zimmermann defines MIMOLA as a language to support high-level-synthesis [3.15]. The language includes advanced concepts such as mechanisms for design by correctness-preserving transformations. Its syntax is somewhat unusual (postfix). P. Marwedel starts to write software for MIMOLA on a teletype. Target architectures are of a VLIW type.
1979	First truly international publications [3.16, 3.8]
1980	G. Zimmermann moves to Honeywell, Minneapolis and starts to use tools built around the language. At Kiel (Germany), researchers led by P. Marwedel start to work on a new version of the language, being based on PASCAL. Also, work on using the language for a retargetable compiler starts. The new tool-set is called MSS2.
1981/84	The work on the first retargetable compiler is published [3.9]
1986	Work on a second generation high-level synthesis tool is published [3.10]. Work on hierarchical test generation is published (more recent version: [3.5]).
1987/89/97	Work on second-generation compilation is published [3.14, 3.11, 3.7].
1987	Tool developers are able to use modern workstations instead of a mainframe. The ported version of the tools is called version 3.45. This stable version is transferred to some academic institutions .work on MIMOLA version 4.0 starts.
1990	Members of the design team move from Kiel to Dortmund (Germany). New tools are designed to accept VHDL or MIMOLA (versions 4.x).
1993	Retargetable code generation becomes a hot topic for DSPs and ASIPs.
1997	MIMOLA is used as a hardware description language for the third generation retargetable RECORD compiler [3.6]. Further use of MIMOLA comes to an end.

2. *Tools*

The design of MIMOLA was mainly driven by its use for architectural and micro-code synthesis. In architectural synthesis, the main input consists of the behavior to be implemented (see Fig. 3.1 (a), top left). For MIMOLA, this behavior is described in a PASCAL-like syntax. Additional inputs include information about structural elements (information about available library components and possibly predefined (partial) structures). Finally, there can be hints for linking behavior and structure.

System behavior is also an input for micro-code synthesis (see Fig. 3.1 (b)). In contrast to architectural synthesis, the structure is fixed and the tool has to compile the system behavior using this structure as the target for the generated binary code (this is the code for the lowest programmable level, which can be either machine code or micro-code). MSS-tool-sets also include simulators for simulating both at the behavioral and at the structural level. Finally, tools for generating self-test programs have also been designed [3.5, 3.2]. This variety of tools allows for smooth transitions between different design tasks [3.13].

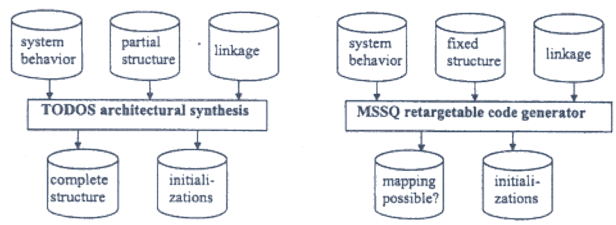

Fig. 3.1 (a) TODOS architectural synthesis
(b) Retargetable machine code synthesis

3. *Salient features of MIMOLA*

A language which supports the mentioned tools has to have description capabilities for (partial or complete) structures, for the required system behavior and for links between these two. In the following, we will describe how MIMOLA provides these capabilities. We will use version 3.45 of the language [3.3], the version for which most of the tools were written.

3.1 Description of system behavior

Ease of learning was a major design goal for MIMOLA. Users try to apply what they already know. Hence, we tried to be consistent with PASCAL. Only a few changes are required to translate a PASCAL program into a MIMOLA program. Fig. 3.2 (a segment of the diffeq high-level synthesis benchmark) gives an impression of the style of MIMOLA behavioral system descriptions.

```
PROGRAM diffeq IS                    (* system behavior       *)
 VAR three,five,a,x,y,dx,u,u1 : (15 : 0);   (* bitvector of bits 15..0 *)
 VAR a,b,c : integer;
 BEGIN ...
  WHILE x < a DO
  BEGIN
   x := dx + x;
   u1 := u * dx;
   y := y + u1;
   u := u - ((u1*(x*5))-(dx*(y*3)));
   ...
  END;
 END;
```

Fig. 3.2 MIMOLA program to be compiled.

For MIMOLA, the only built-in data type is the vector of fixed set of four logic values (0, 1, X, Z). Vectors of these values can be used throughout the language. These vectors have descending index ranges and are denoted by "(upper-index : lower-index)". See the definition of variables in Fig. 3.2. In contrast to VHDL, all data elements are described in terms of such vectors. There is no special data type for single bits: a single bit is just a vector with a single element. Due to this, descriptions closely resemble the actual hardware structure. Also, arguments and results of arithmetic functions are always bit-vectors. There is no need to consider arithmetic functions with various combinations of argument types. This is important in order to avoid combinatorial explosion of the number of predefined functions. By default, all arithmetic operators assume that their arguments have to be interpreted as twos complement numbers. In cases where this is not the intended interpretation, special operators have to be used. For example, operators like |>| interpret bit vectors as natural numbers. For a behavioral description such as the one in Fig. 3.2, the MIMOLA tool-set can synthesize custom hardware and it can compile it into the code of a given target machine, described by its structure (more recent compilers [3.6] also accept instruction set descriptions of target machines).

3.2 Description of system structure

As an example, we consider the simple processor in Fig. 3.3 and its textual description in Fig. 3.4.

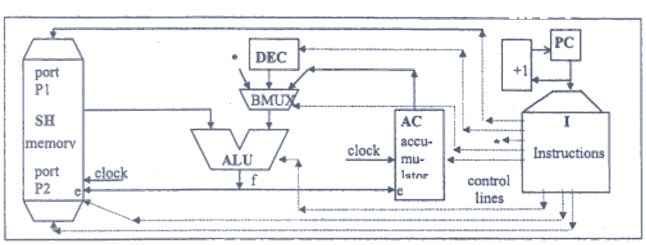

Fig. 3.3. Simple processor.

The description starts with introducing a name for the current design object (simplecpu). The body is structurally described (keyword STRUCTURE). The structural body defines all available components (parts). Each part is of a certain library element type, called module. To make things easier for the user, the syntax of modules resembles that of PASCAL procedures.

Module bodies, may be described either behaviorally or structurally. The example contains a behavioral description of module Balu. Each case line describes an operation mode of the component. Case labels denote control codes which are required to select an operation mode.

Behavioral modules of local modules cannot contain anything but a set of concurrent signal assignments or assignments to local variables. Only a single assignment is allowed for each of the outputs and local variables. This approach allows a very elegant description of multi-port memories.

```
MODULE simplecpu;
 STRUCTURE of Simplecpu IS
  TYPE
   word = (15:0);
  PARTS
   ALU : MODULE Balu(IN a,b:word; FCT s:(1:0); OUT f:word);
       BEGIN
        f <- CASE s OF      (*signal assignment*)
            0: a * b;
            1: b;
            2: a + b;
            3: a - b;
           END
       END;
   SH : MODULE S64k ..(IN e: word; ....
   I  : MODULE SRAM ..
   PC : MODULE Rw  ..
   AC : MODULE REG  .. (IN e: word; ....
  CONNECTIONS
   ALU.f -> SH.P2.e; ALU.f -> AC.e; ...
 END_structure;
```

Fig. 3.4. MIMOLA processor description.

The approach is general enough to specify arguments and control codes involved in a component operation. Simpler approaches exist, but are definitely not sufficient. On the other hand, more general approaches (e.g. arbitrary behavioral descriptions) could not be used by any of our three main tools.

The ability to describe library components and system behavior in the same language has always been one of the strong points of our approach and has been exploited by our tools.

Nets can be described in a CONNECTIONS-definition which contains sources and sinks for the nets.

3.3 Links between behavioral and structural elements

Synthesis is usually not a fully automatic process, but relies on user guidance. Such guidance is frequently provided through the help of control files or pragmas (pseudo comments). Since MIMOLA is a synthesis-oriented language, language elements for user-guidance can be built into the language. As a result, there is a reduced risk of inconsistencies.

Major emphasis of the tools centered around the language is on the support of programmable instruction set architectures (e.g. ASIPs, core processors). For these, some of the registers and memories serve a special purpose: for example, a register is used as the program counter, a certain memory is used to hold instructions and so on. Tools usually cannot figure out, which of the registers and memories serve

a certain purpose. Therefore, we add hints to the descriptions. These are introduced by the keyword LOCATIONS.

```
```

Furthermore, MIMOLA is also able to represent program transformation rules explicitly. Program transformation rules provide a means for transforming the behavioral specification before generating an implementation. According to our knowledge, MIMOLA is the only HDL including this feature. Designers usually have some clever ideas about essential elements of the design. It would be silly not to take advantage of the designers knowledge. In an earlier paper [3.8], we have demonstrated the effect of such knowledge on the efficiency of the resulting design. MIMOLA contains several language features which facilitate capturing the designers knowledge. These consist of features for

Manual operation to operator binding

Designers are frequently able to provide valuable hints about which hardware operator should be used to perform certain operations. Such hints can be included in MIMOLA descriptions.

```
a +_alu b      (* for + use alu *)
```

Manual variable to storage binding

MIMOLA provides an extension to PASCAL to indicate such a binding. Example: Assume SH is a memory. A variable called zero can be bound to location 0 of this memory by the declaration:

```
VAR zero: word AT SH[0];
```

Manual operation to control step binding

Towards this end, MIMOLA provides special strictly sequential blocks. Such blocks contain parallel blocks, each of which describes the operations in a control step. Strictly sequential blocks are excluded from automatic scheduling.

3.4 Initialization

```
INIT SH[0..20]:=0;
```

Several of our tools generate requirements for the initialization of memory locations. For example, our retargetable code generator basically just generates such requirements, called † binary code. It is desirable to store these requirements independently of structural descriptions. Therefore, we have created a language element for it.

```
INIT SH[0..20] : = 0 ;
```

4. Further information

The preceding examples were based on version 3.45 of the MIMOLA language. Reprints of the corresponding language manual are available [3.3]. Cleaning-up the language has resulted in a more comprehensive document [3.1]. Papers describing tools using MIMOLA published from 1989 onwards can be accessed through the world-wide web (http://ls12-www.cs.uni-dortmund.de).

MIMOLA References

[3.1] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer, *MIMOLA reference manual -version 4.1-*, //ls12.www.cs.uni- dortmund.de

[3.2] U. Bieker and P. Marwedel, "Retargetable Self-Test Program Generation Using Constraint Logic Programming," *32nd Design Automation Conference*, 1995

[3.3] R. Jöhnk and P. Marwedel, *MIMOLA reference manual -version 3.45-, Technical report 470, Computer Science Department*, University of Dortmund, 1993

[3.4] D. Ku and G. De Micheli, *Hardware C - A language for hardware design, version 2.0, Technical report CSL-TR-90-419*, Stanford University, 1990

[3.5] G. Krüer, "A tool for hierarchical test generation," *IEEE Trans. on CAD*, Vol. 10, pages 519-524, 1991

[3.6] R. Leupers, *Retargetable Code Generation for Digital Signal Processors, Kluwer Academic Publishers*, 1997

[3.7] R. Leupers and P. Marwedel, "Retargetable Code Generation Based on Structural Processor Descriptions," *Journal on Design Automation of Embedded Systems*, 1997

[3.8] P. Marwedel, "The MIMOLA design system: Detailed description of the software system," *16th Design Automation Conference*, pages 59-63, 1979

[3.9] P. Marwedel, "A retargetable compiler for a high-level microprogramming language," *ACM Sigmicro Newsletter*, Vol. 15, pages 267-274, 1984

[3.10] P. Marwedel, "A new synthesis algorithm for the MIMOLA software system," 23rd *Design Automation Conference*, pages 271-277, 1986

[3.11] P. Marwedel and L. Nowak, "Verification of hardware descriptions by retargetable code generation," *26th Design Automation Conference*, pages 441-447,1989

[3.12] P. Marwedel and W. Schenk, "Improving the performance of high-level synthesis," *Microprogramming and Microprocessing*, Vol. 27, pages 381-388, 1989

[3.13] P. Marwedel and W. Schenk, "Cooperation of synthesis, retargetable code generation and test generation in the MSS," EDAC-EUROASIC93, pages 63-69, 1993

[3.14] L. Nowak, "Graph based retargetable microcode compilation in the MIMOLA design system," *20th Annual*

Workshop on Microprogramming, pages 126-132, 1987.

[3.15] G. Zimmermann, *Report on the computer architecture design language MIMOLA*, Technical Report 4/77, Institut für Informatik & P:M., Universität Kiel, 1977

[3.16] G. Zimmermann, "The MIMOLA design system: A computer-aided digital processor design method," *16th Design Automation Conference*, pages 53-58, 1979

IV. The Hardware Description Language DACAPO III

Contributed **by Franz J. Rammig**, Universität Paderborn, FB Mathem./Inform., Paderborn, Germany,
and **Christoph Ohsendoth**, Universität Dortmund, FB Informatik, Germany

1. Historical notes

The HDL DACAPO III originated from an evolutionary process that started in 1977. In the beginning this language has been called CAP, standing for Concurrent Algorithmic Programming. The name reflected several basic design principles that have been kept during the entire language evolution:

- CAP was intended as an algorithmic language, covering the abstraction level above the RT-level. At the same time it was designed in such a way that lower levels (gate and RT) have been covered as well.
- CAP was intended to express a high degree of concurrency in an easy and concise manner.
- The design of hardware was looked at as a special kind of programming. At this time (prior to the age of synthesis) this was a very unusual point of view.

The semantics was formally defined in terms of interpreted Petri Nets. We implemented a compiler, a simulator and a generator for microprocessor based controllers from specifications given in CAP.

In 1979, a revised version of the language then was called CAP/DSDL [4.6, 4.1]. We added a couple of new principles, including an assertion mechanism and interpreted abstract data types. CAP/DSDL has been used by SIEMENS in a couple of projects and a subset, named FBDL [4.2], became part of SIEMENS' VLSI design system VENUS [4.5].

In 1985 a third redesign of the language took place. Now concepts of modularization, inherited from MODULA and advanced generic concepts were added. This language was called DACAPO III [4.9]. A software system consisting of a compiler (analyzer), a high performance (mixed compiled mode and event driven) simulator [4.4], a test pattern generator and an intelligent result generator has been implemented and marketed by DOSIS GmbH. Later a parallel implementation of the DACAPO III simulator has been implemented on Transputer systems [4.7]. DACAPO III has been licensed by companies like Nixdorf Computer AG and played its role in a couple of research projects.

2. Some basic concepts of DACAPO III

2.1 Basic Notations

DACAPO III is a language that looks like MODULA II as far as possible. Concerning variables DACAPO makes the distinction between storing (called explicit) and non storing (called implicit) data carriers. The latter ones need a continuous data assignment to be defined while the first ones keep a once assigned value as long as no other value has been assigned to. In addition so called auxiliary variables are available, comparable to variables in VHDL. DACAPO follows the idea of orthogonality concerning operators and operands (overloading). So all operators are not only defined on scalars but on arrays and records, too. Most aspects where DACAPO is different from MODULA lay in a domain outside the syntax of MODULA, while aspects covered by MODULA have the same meaning in DACAPO. The real power of DACAPO, however, originates from the extensions beyond MODULA.

2.2 DACAPO III at the Algorithmic Level

The algorithmic part of a DACAPO description consists of a single compound statement. It may contain other statements, including compound statements. There are four different types of compound statements:

- sequential: seqbegin S1;...;Sn end;
- concurrent: conbegin S1;...;Sn end;
- parallel: parbegin S1;...;Sn end;
- compact: begin S1;...;Sn end;

The semantics of the compact compound statement are equivalent to those of the compound statement of languages like MODULA: The embedded statements are executed sequentially and uninterruptable in the given order. Slightly different is the semantics of the sequential compound statement. In this case, too, the embedded statements are executed sequentially in the order written down, but the consecutive execution is no longer guaranteed. In fact the statements S1,...,Sn now may be seen as (lightweight) threads. When a concurrent compound statement is initiated, all embedded statements are initiated concurrently. It is finished when the last one of the embedded statements has been executed. This means that (lightweight) threads S1,...,Sn are created and initiated. Contrarily to this asynchronous interpretation, the embedded statements of the parallel compound statement are executed in a strictly synchronized manner without any interdependencies.

A little example may illustrate some of these concepts:

```
conbegin
        seqbegin
                a:=1;
                b:=10/a;
        a:\0;
        end
```

In this little fragment a concurrent compound statement has two embedded statements, the first one being a sequential compound statement. Due to the asynchronous characteristic of the concurrent compound statement and the non-atomicity of the sequential compound statement it is not excluded that the variable a gets a value 0 before the assignment b := 10/a is executed. Replacing the sequential compound statement by a compact one would overcome this problem.

Other statements of algorithmic DACAPO III are similar to MODULA: assignment, *while, repeat, if, case*. The for-statement has a slightly different semantics: It is interpreted as a shorthand of a compound statement.

Usually in imperative languages the flow of control is entirely given by the control structure of the algorithm. This makes it difficult to describe the synchronization of the algorithm with externally given events (e.g. keystrokes or clock signals). For this purpose in DACAPO any statement may be prefixed with *at event do*. The semantics is, that after the prefixed statement is ready for execution due to the normal control flow, the event mentioned in the prefix has to occur to initiate the statements execution.

Procedure and function calls are handled as in MODULA, with slight differences in the parameter passing mechanism. An important difference is the introduction of timing. Assignments and empty statements may be delayed. The delay may be constant or calculated dynamically at runtime. The basic delay model is transport delay. More complicated delay models can be programmed easily (see example below).

```
c:= a delay (if a>b then loadtime/a else prechargetime+b)
```

It has to be noticed that in DACAPO an assignment statement with an associated delay consumes time, i.e. not only the assignment itself but also the initiation of a statement that follows sequentially is delayed.

2.3 DACAPO III at the System Level

At this level support of modularization, abstract data types (structural object orientation), interrupt handling and generics are of interest. The entire algorithmic power of DACAPO is a very useful at the system level.

2.3.1 Modularization

A DACAPO description is composed of modules. Like in MODULA there are *definition modules* to specify the interfaces and *implementation modules* to specify the internals. Modules may be further organized using procedures, functions and export procedures. But contrarily to languages like MODULA procedures, functions and export procedures may be defined as types. This allows to generate an arbitrary number of instances of such an object,, even complex structures like multidimensional arrays of procedures.

2.3.2 Generics

Procedures, functions and export procedures are the level of granularity to apply generics in DACAPO. Each type and each constant used within such a construct may be a generic one that may be personalized individually at each instantiation.

2.3.3 Abstract Data Types

Above, the term export procedures has been used several times. The characteristic property of an export procedure is the export of the operations (methods) for the manipulation of its internal data instead of the data itself. The declaration looks like the following example:

```
type processor = export(instr1,...,instrn) procedure
processor: ... end;
```

Export procedures allow to describe implemented ADTs, i.e. ADTs where the defining equations have been replaced by implemented code. In DACAPO for each ADT method a procedure must be provided. Internally the methods are defined based on an internal carrier structure, that may be an ADT as well.

The following example defines a fifo queue as generic type and instantiates three instances: one fifo of 64 single bytes (called byte-fifo-array), one consisting of four records, each of them containing two words of different length (called record_fifo), and finally one with the capacity of two arrays of 16 words each (called array_fifo).

```
type fifo =                                  «type definition»
generic const depth ; type item_type:
export ( reset, insert, remove) procedure fifo ;

var buffer : array [ 0 : depth-1 ] of item_type ;
next, first : bit (depth) ;

procedure reset ;
seqbegin
first, next := 0
end ; {reset}

procedure insert ( in item : item_type ; out full : bit ) ;
conbegin
if (next |+| i0001î ) mod depth = first
then full := ìîî
else conbegin
buffer [next] := item ;
next := (( next |+| i0001î ) mod depth ) ;
full := ì0î
end
end ; {insert}
```

1988.

[4.8] A. Oczko, "Hardware Design with VHDL at a Very High Level of Abstraction," *Proc. 1st European Conference on VHDL*, Marseille, 1990.

[4.9] C. Ohsendoth and B. Reusch, "System Level Description and Simulation With VHDL and DACAPO-III," *Proc. European Simulation Multiconference*, Rome, Italy, 1989.

V. Ten Years of ELLA

Contributed by, **John D. Morison** and **Cleland O. Newton**, Defense Research Agency, Malvern, U.K.

1. History

The ELLA language is the result of more than 10 years experience in modeling and design of digital electronic systems. The project started in 1979 at the UK Defense Research Agency (DRA) at Malvern, then known as RSRE, and a license for the first prototype system was sold to a UK company in 1982. In 1985 the system was marketed commercially, while the language evolved steadily in an upwards compatible way. ELLA has been used to design hundreds of circuits from those of a handful of gates to complex VLSI systems of over 500,000 gates.

The project is now complete with the definition and implementation of a final version of the language, 'ELLA 2000', described in a book [5.1] and available in the public domain in source code and executable forms [5.2].

2. Underlying model

ELLA is designed to be extremely safe to use, with a simple underlying model close to the network view of hardware. The user thinks in terms of creating and connecting together blocks of hardware rather than handling processes and simulator events. This 'structural' view is often looked on as low level, as opposed to a 'behavioral' view which is looked on as high level. Possibly the main contribution of ELLA is to demonstrate that a 'structural' view when taken to a high enough level can be just as abstract as other approaches. The underlying 'structural' model, as well as being close to the real world of hardware and hence appealing to the engineer, is also well suited to efficient simulation, program transformation (one third of the language is defined in terms of transformations into the remainder), synthesis and formal methods. The formal semantics of a core subset has been defined [5.3].

Distinguishing features

3.1. Distinction between circuit parameters and signal values

In ELLA the values used for circuit construction and those used for signal description are disjoint. The circuit is fully defined before the start of simulation, preventing signal dependent hardware. Construction generics allow integers, signal types and functions (which may themselves be parameterised), to be the parameters of generic functions. The disjointness is achieved by insisting that all signal values must be tagged, leaving untagged values for hardware construction.

For example:
TYPE integer = NEW i/(0..255).
defines a new signal type called 'integer' with values from i/0 to i/255. These signal values are disjoint from numbers used to define circuit size:
FN TRANSPOSE = ([3][4] integer : matrix) -> [4][3] integer: [INT p=1..4][INT q=1..3] matrix [q][p].
defines a function which may be instantiated by:
LET new_matrix = TRANSPOSE(my_matrix).
TRANSPOSE could also be described generically to work on all sizes of matrices.

3.2. No distinction between behavior and structure

It is generally accepted that behavior refers to simulation and structure to implementation. In many HDLs behavior is described using software constructs, while structure is described using net-lists. For example VHDL uses a dualism of concurrent processes (software constructs) and component instantiations (net-lists). To make simulation possible a behavior needs to be associated with a structure, so structure is elaborated as the flattening of a hierarchical description of behavior. ELLA bases its descriptions on net-lists, using a form which is exactly consistent with the functional expression of behavior. Software constructs compatible with structure are supported (for example sequential assignment is supported but not GOTO). Constructs in ELLA thus express behavior and structure at the same time.

The following example shows four ways of creating the structurally identical cascade of three AND gates in ELLA, showing 'structural', 'behavioral', 'mixed' and 'recursive' styles:

```
- (1) - MAKE AND : and1 and2 out.
JOIN (in1,in2)->and1, (and1,in3)->and2, (and2,in4)->out.
- (2) - LET out = in1 AND in2 AND in3 AND in4.
- (3) - VAR out := AND(in1,in2);
out := (MAKE AND : and.
JOIN(out AND in3, in4)->and.
OUTPUT and);
- (4) - LET out = CASCADE {AND,4} (in1,in2,in3,in4).
where CASCADE is a generic macro defined for Boolean signals by:
MAC CASCADE {FN GATE = [2]bool->bool,INT n} = ([n]bool: input) -> bool:
IF n=1 THEN input[1]
ELSE GATE (input[n], CASCADE {GATE, n-1} input[1..n-1])
FI.
```

Here AND is a user-defined function, not a language-defined primitive. It could for example define the complex number addition of row vectors (which would require a change to bool in the definition of CASCADE).

3.3. A universal unknown value

ELLA offers a built-in unknown value for every signal type, with strictly defined semantics for each language construct. Each unknown value, rather than being an additional value of the type, represents the set of all possible values and therefore cannot be specifically tested for in a multiplexing operation. In contrast VHDL provides the leftmost value as the unknown value.

3.4. Variant record types

ELLA offers variant records in the form of tagged unions, which are particularly valuable for trapping design faults:

```
TYPE int = NEW i/(0..1024),
code = NEW (add | sub | jump | stop),
opop = NEW (address1 & int | address2 & (int,int) | instr & code).
```

3.5. A specific backwards-looking time primitive

In ELLA the model of time employs signal history to define a new set of signal values. The delay primitive has to be wrapped up as a simple function before it can be used.

```
FN DELBOOL = bool -> bool : DELAY (false, 1).
```

Here 'false' defines the initialization value of the signal at time zero and 1' the number of delay cycles. An ELLA simulator may be expected to use an event driven algorithm but any such mechanism is hidden from the user. ELLA's simple model of time makes it naturally suited to synchronous design because the simulator provides an automatic clocking mechanism. Alternatively the user may associate the ELLA delay unit with say 1 nsec or with asynchronous events. Undelayed feedback is not illegal in ELLA but is discouraged; if it causes oscillation the unknown value is substituted automatically by the simulator.

3.6. A language and intermediate format designed for transformations

The ELLA system is designed to allow language transformations. This has enabled the language to be extended to progressively higher levels without change to the simulator. New constructs such as bi-directional signals and complex generics are automatically transformed to the primitive set of constructs of the simulator. Tools are provided to write special purpose transformations.

Specific transformations have been designed for synthesis, such as mapping integers to Boolean. Both the parse-tree format of the database and the transformation writing tools are available to end users as a C language toolkit.

4. Other design features

ELLA has been designed with conventional virtues in mind. Safety is achieved by strong type checking, lack of overloading and compile time checks. Language constructs are orthogonal, i.e. minimal but usable with the same meaning wherever relevant. BIOPs offer a full repertoire of arithmetic, character and bit operations, sufficient for the user to specify his own floating point algorithms. The ELLA language is designed to work within an environment, which comprises a library and context system for design management, a transformation friendly database and an interface to a simulator and other tools.

5. Strengths

- ELLA's simple hardware-based model is attractive to engineers because it is easy to relate an ELLA description to the hardware equivalent.
- ELLA is attractive to education because it makes possible a logical progression of tuition from standard network descriptions through functional expressions to register transfer level descriptions and beyond.
- ELLA is attractive for design particularly for digital signal processing applications, because of its simplicity, functionality, inherent concurrency and powerful generics.
- ELLA is attractive for synthesis because of the transformation facility, which includes the ability to transform to VHDL, and because any design automatically has a hardware equivalent.
- ELLA is attractive for formal methods because it has a small tractable core subset with a formally defined semantics. Any ELLA design may be transformed into this subset and hence is amenable to a range of formal methods [5.3].

6. Weaknesses

- ELLA has not been considered for standardization at a time when there is universal demand for standards.
- Compatibility between ELLA and VHDL is low at high level because of the differences described above. The ELLA to VHDL translator transforms out high level ELLA constructs and therefore generates low level VHDL.
- ELLA has been seen in the UK as a proprietary system and is little known outside the UK.
- The only full implementation of ELLA is in Algol. Although the Algol code is machine translated to C, the system is not easily portable.

7. Use of ELLA

Use of ELLA peaked at over 100 licensees, with more than 100 commercial designs completed. At least one was at the

500,000 gate level, with several around 50,000 gates. These have been genuine high level designs employing user-defined abstract types. Designers went on record to claim right-first-time operation and to say that circuits of such complexity would not have been attempted without it. Direct sales of ELLA tools exceeded 1.5M. Commercial interest in ELLA has now lapsed but interest in universities survives.

ELLA References

[5.1] J.D.Morison and A.S.Clarke, *ELLA 2000*, McGraw-Hill, 1993, 1995.

[5.2] ELLA public domain software FTP address: src.doc.ic.ac.uk:packages/ELLA.

[5.3] H.Barringer, G.Gough, B.Monahan, and A.Williams, "A Process Algebraic Semantics for Core ELLA," University of Manchester Technical Report UMCS-93-2-1, 1994.

VI. The ART* System

Contributed by **P. Prinetto**, Politecnico di Torino, Dipartimento di Automatica e Informatica, Torino, Italy

1. Introduction

The **ART*** is a register transfer level simulation system partially implemented at the Politecnico di Torino, Italy, in the early 1980s [6.2]. The system has been built mostly resorting to the **RTS1a** simulation system, previously developed by H.-J. Knobloch at the Institut für Datentechnik of the Tech. Hochschule Darmstadt, Germany [6.3], [6.4]. The architecture of the overall system is shown in Fig. 6.1, where four main blocks are distinguishable: the *HDL Compiler*, the *Unit Management System*, *Waveform interpreter*, and the *Simulator*.

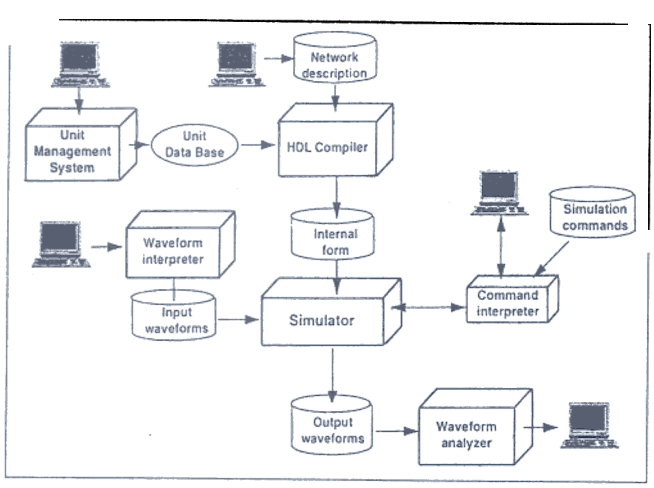

Fig. 6.1. Architecture of the ART* system

2. The HDL

The ART* HDL is strongly typed, non-procedural, applicable at RT and logic levels. It is rather flexible, free of inconsistencies, and not restricted to a specific technology or design method. Referring to the classification of [6.1], Fig. 6.2 outlines the covered abstraction levels and representation domains.

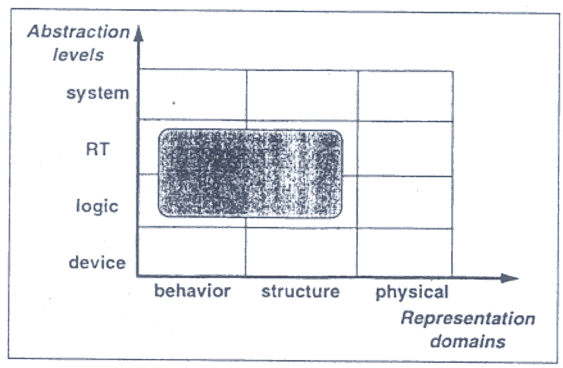

Fig. 6.2. ART* HDL applicability.

The main characteristics of the ART* HDL may be summarized as follows:

- The underlying model in which system behavior is interpreted is a **Mealy automaton**:
  - all the alphabets (input, output, state) consist of ordered sets of Boolean values (0-1, true-false, high-low, etc.)
  - the timing model is very simple (before/after relation), thus avoiding the mixture of synchronous and asynchronous behavior.
- A great deal of flexibility is provided to the designer: the same design may be described in several equivalent ways, in dependence of the aspects he is mainly concerned with:
  - many constructs are available to describe complex automata in a very compact and easily understandable way, suppressing repetitive pieces of text
- Design hierarchy and partitioning can be exploited through subsystems described at different abstraction levels (*units*, like in CASSANDRE). Such subsystems may be notated by a set of statements to be treated as a procedurein behavioral descriptions, or to be treated as a building block in structural ones.. The scope of internal variables is similar to Pascals. The user may declare his own units and/or utilize some previously defined ones stored in a dedicated Unit Data Base.

An example is given in Fig. 6.3, in which two alternative ways are given to represent a multiplexer, the former being conventionally referred to as *structural(a)* and the latter as *behavioral* (b), respectively.

```
-INPUT- A[1:4], B[1:4], S;
-OUTPUT- Y[1:4];
-BEGIN-

Y = ( A ; B ) [ S ]

-FINISH-
```

(a)

```
-INPUT- A[1:4], B[1:4], S;
-OUTPUT- Y[1:4];
-BEGIN-
-IF- S' -THEN- Y = A
        -ELSE- Y = B
-FI-
-FINISH-
```

(b)

Fig. 6.3. Alternative descriptions of 157 (quadruple 2:1 multiplexer without STROBE signal).

```
        SN74163 : SYNCHRONOUS 4-BIT UP-COUNTER
        behavioral description
        one clock cycle > two simulation cycle      "
-INPUT-         CLEAR, ENP, ENT, LOAD, D, C, B, A, CLOCK;
-REGISTER-      QD, QC, QB, QA;
-TERMINAL- RIPCAR;
-BEGIN-
RIPCAR=QD*QC*QB*QA*ENT,
-IF- CLOCK -THEN-
-IF- CLEAR'
        -THEN-                  "clear = 0"
        (QD,QC,QB,QA) <= #0B4
        -ELSE-
        -IF- LOAD'
        -THEN-                          "clear = 1, load = 0"
        (QD,QC,QB,QA) <= (D,C,B,A)
        -ELSE-                  "clear = 1, load = 1"
        -IF- ENP*ENT
        -THEN- (QD,QC,QB,QA)<=INC((QD,QC,QB,QA))
        -FI-
        -FI-
-FI-
-FI-
```

Fig. 6.4. A sample ART* HDL description.

3. The ART Simulator*

The ART* Simulator enables the user to simulate the behavior of a digital system whose description has been previously translated by the HDL Compiler into an internal tabular form (Fig. 6.1).

The simulator cyclically changes between two modes of operation: INTERPRETE and SIMULATE. In the former it accepts any sequence of control commands either from a set of command files or from the user terminal. As soon as the *Resume* command is read, the SIMULATE mode is entered and the simulation started. When an error is detected or a break condition becomes true, the INTERPRETE mode is re-entered.

As previously stated, the ART* system models hardware behavior by a Mealy automaton. This simulator was one of the most efficient in the late 70's. It is **table driven** and the elemental time unit of the discrete time scaling is one simulation cycle. The default behavior of the simulator suffices to model single phase clock systems. If more complex clocking schemes are required, explicit clock signals must be introduced. To ease the design verification task, many facilities are provided, the most significant ones being:

* forcing the system to any initial state condition
* imitating the behavior of the environment of the system by supplying primary inputs with arbitrary patterns. Input waveforms may be specified through an ad-hoc *Waveform Description Language*, which provides facilities to describe peculiar behaviors, such as: periodic, random, increment, decrement, count, etc. (Fig 6.1)
* inserting break points, specified by various activation conditions. Such conditions are nowadays common on most simulation environments, but were innovative and peculiar at that time.
* tracing the simulation steps.
* saving the status of the system either at a given cycle or periodically, and later on restoring it.

4. Conclusions

The **ART*** and the **RTS1a** systems have been widely used in both academic and industrial environments, thanks to a complete set of reference manuals and collection of examples, distributed with the software.

In addition, **libraries** including the descriptions of most MSI-LSI chips of the TTL family and a bipolar microprocessor family were made available.

Although the simulation systems presented some intrinsic limitations, mainly due to the underlying timing model and to the applicability, restricted to register transfer and logic level descriptions, they provided some innovative features that became common in systems of the next generation, only.

ART References

[6.1] R. Camposano, High level synthesis: a tutorial, IEEE Design & Test of Computers, October 1990.

[6.2] S. Gai, M. Mezzalama, P. Prinetto, and F. Somenzi, ART* : a register Transfer simulation system, IEEE ICCAD-83, September 1983, Santa Clara, CA (USA).

[6.3] H.-J.Knobloch, RTS Ia ein System zur formalen Bescreibung und Simulation komplexer Schaltwerke, Dissertation, Technische Hochschule Darmstadt, 1978.

[6.4] H.-J.Knobloch, Description and simulation of complex digital systems by means of the register transfer language RTS 1a, In P.Antognetti, D.O.Pederson, and H.De Man (Hrsg.),*Computer Design Aids for VLSI Circuits*, pp. 285-320, Sijthoff and Noordhoff, NATO Advanced Study Institute Series E-48, 1981.

7. The CASCADE and its multi-level mixed mode simulator

Contributed by D. Borrione, Cl. Lefaou, J. Mermet, Laboratoire TIMA, Grenoble, France

1. Introduction

The CASCADE project started on 1980 was the continuation of a research started in the sixties at IMAG. Initial studies were aimed at defining the concepts for hardware description, and the basic mechanisms for building simulation software, at a given level of design. At that time simulation was a research topic, and far from being recognized as a valuable design aid tool. Over the years, languages were defined, and simulators implemented, for a variety of modeling levels:

CASSANDRE [7.7] at register transfer level,
IMAG[7.5] at electrical (circuit) level,
LASCAR [7.1] at functional architecture level,
LASSo[7.2] at system level.

All these languages were independent.

Our participation in the CONLAN effort, and the emerging need for multi-level modeling tools and simulators, motivated research with the main objective of integrating all description levels (discrete and continuous) into a single language, allowing mixed level modeling, and mixed mode simulation. CASCADE was the first language to achieve and implement totally this goal.

CASCADE covers all hardware modeling levels, from the abstract system behavior down to the electrical behavior of basic components. It is worth saying that the most advanced language to-day -VHDL- still does not offer system level primitives, nor analog concepts (VHDL-AMS in the IEEE standardization process will eventually provide it).

More specifically, CASCADE can be divided into:

- core CASCADE, a kernel of notions, data types, operators and constructs, common to several or all levels of abstraction, with unique syntax and semantics.
- predefined language levels, which include the applicable value types, carrier types, operators, primitive hardware modules, and statements associated to a primitive control model if any.

CASCADE is strongly based on CONLAN [7.3, 7.8]: all CONLAN notions related to the genericity of carrier types, parameterization of descriptions, user definition of segments as a means of expressing modularity in structure and behavior have been incorporated. As with CONLAN, CASCADE is a strongly typed language, which accepts *user-defined types*, and performs extensive type checking in expressions and procedure/function calls. Except for a few minor changes, the syntax of CASCADE follows the syntax of CONLAN when referring to the same statement.

The most significant differences between CASCADE and CONLAN are that CONLAN provides toolmakers with digital primitives for defining user languages, CASCADE is only oriented towards circuit designers,. Furthermore it does not limit itself to discrete modeling.

Thus, none of the toolmaker specific statements of CONLAN has been incorporated in CASCADE: new types may only be defined as subtypes of existing ones; language segments may only extend their reference language by adding new types, functions, procedures and descriptions, without any syntax alteration. Furthermore extensions to CONLAN were made to incorporate electrical and mixed discrete-continuous modeling: the value type REAL, the electrical node and electrical wire carrier types which hold voltages and currents and the derivation operator for writing differential equations. The intercommunication of modules described at various CASCADE levels was made possible even if their interface carriers are of different types (default conversion functions are provided, which may be overridden by user-defined conversion functions).

Therefore, CASCADE was not *yet another hardware description language* . It was the synthesis of over 15 years of efforts by several groups of researchers.

2. Modularity

In CASCADE two kinds of decomposition exist:

1- Structural decomposition ,which corresponds to physically disjoint parts. The design is described as a hierarchy of networks of interconnected boxes modeled by the description segment.

2- Functional decomposition which is modeled, as in programming languages, by function and procedure segments and corresponds , in a given box, to various operations on the same resources.

2.1. Description

A description segment is the model of an arbitrarily complex hardware module. It can have generic parameters, thus representing a *family* of modules of similar structure or behavior. A particular circuit is an instance of a description segment, where generics have been fixed.

A description, being a full model, can be verified independently of any model in which it is instantiated. Therefore, access to global variables is forbidden (with one exception at the electrical level where *shared read-only* variables are used). A unit communicates with its enclosing environment only through its interface carriers typed as in (input), "out" (output), inout (alternatively input or output), or nd (non-directional, reserved for switch and electrical levels).

2.2. Description segment

In CASCADE (as in CASSANDRE or VHDL) a description segment can be structural, functional, or a mixture of both (except in POLO or CASTOR which are only structural levels). It has two optional parts:

• the structural part contains local carriers and constants

(declare statement), the naming of enclosed units (components) and their permanent interconnection (use statement).

- the functional part, introduced by key-word relations, describes the behavior . The statements and primitives available depend on the stated language level of CASCADE.

3. Behavior modeling and abstraction levels

A CASCADE language level is defined by:

1- a set of primitive value and carrier types, and a set of operations on the objects of these types;
2- a (possibly empty) set of primitive function, procedure and description segments;
3- a (possibly empty) set of statements available for writing relations.

3.1. Value and carrier types

The user may define subtypes of primitive value types by enumeration of the subtype elements, or by stating the subtype characteristic function.

As in CONLAN, a carrier type is derived from a generic type (which defines the assignment operators, the time behavior and memory properties.). Actual carrier types may be renamed by the user. The most widely used carrier types are given an abbreviation in each CASCADE language level. The very powerful ARRAY generic type of CONLAN applies to value and carrier types: any number of dimensions; increasing or decreasing bounds; selecting an element or a slice, one or several times, by indexing; shifts and rotates by indexing; transpositions and reduction operators.

3.2. Lasso: addressing the system level

In LASSO, one specifies the timing and functional behavior of digital systems, and the interactions between system components, but no implementation detail. Routing of messages in a network, or synchronizing communicating parallel processes, are typical examples.

The behavior of a LASSO model is based on the separation between operations and control.

The operative part is expressed as assignments to abstract variables which may contain integer, string or symbolic values. Operations may be permanently valid, or may depend on the model control part.

The control part uses a directed *control graph* model, inspired by the LOGOS project [7.5]. *Places* hold Boolean values (carrier type CTRL), and memorize the occurrence of events. *Transitions* modify the value of one or several CTRLs to which they are connected. A transition "firing" takes into account one or more events (its input CTRLs), thus creating other events (its output CTRLs). A variety of transitions model the launching and synchronization of parallel actions ("AND"), switches ("TEST"), decoders ("INDEX"), and selections ("SELECT"). Operation statements may be attached to a transition. When the transition fires, these statements are sequentially executed.

The LASSO simulator is event-driven: a simulation event is a change of value of a CTRL, which may cause a transition to fire. At any point of the simulated time, the simulator first determines all the transitions that may fire, then fires them one after the other, executing their associated operative part. A new cycle is performed if new transitions have become able to fire as a result.

3.3. Lascar And Synchronous CASSANDRE: Addressing the Architecture and Register Transfer Levels

LASCAR and Synchronous CASSANDRE are the architectural and register transfer levels of CASCADE. Time is expressed in terms of clock cycles. The system state may change only when a clock pulse occurs, and combinatorial circuits are assumed to stabilize before the next clock pulse. The behavioral part of a description may represent:

- **Combinatorial circuits** are memory-less. All statements are connections to TERMINALs, with various value types, of possibly complex expressions built on the values of other terminals. All statements are concurrent and permanently active.
- **Sequential circuits** contain memory elements, registers and latches, holding various value types.
- **The control** is modeled by an **automaton**.

Statements that depend on a given state are labeled with the symbolic state name (:IDENTIFIER:); within a state, all operations are concurrent. An implicit state register contains the current state. Sequencing is modeled by loading the state register with a new state value, which takes one clock period. The difference between LASCAR and CASSANDRE is in the operators and value types available (only boolean in CASSANDRE, also arithmetic in LASCAR more oriented towards performance simulation).

The simulatable model is compiled. A stabilization algorithm is applied. All assignments to TERMINALs are statically put in a causal order by the compiler. This guarantees the stabilization of combinatorial parts in one cycle, when no loops are found, in a cycle based simulation. The automaton is compiled as a "case" statement on the value of the state register (see example in CASSANDRE paper).

3.4. Asynchronous CASSANDRE: addressing the register transfer level with another time model

Asynchronous CASSANDRE allows the modeling of transport delays, expressed in terms of time units. Past values of carriers may be referenced in expressions. A synchronization pulse may be the rising or falling edge of a local Boolean carrier and not necessarily an externally provided clock pulse.

3.5. Polo: Addressing the Gate Level

The lower levels of CASCADE (POLO, CASTOR and IMAG_F) provide the designer with a set of predefined description segments, whose behavior is directly implemented in the simulator [7.6].

The primitives in POLO are the usual logic gates, transfer gates, tri-state gates, and a simple model of a unidirectional N inputs 1 output wired-or BUS. Their interface ports are directional. They are typed:

```
VARLOG = VARIABLE (LOGIC4, 'u)
         with LOGIC4 = {'0, '1, 'u, 'z}
```

3.6. Castor: addressing the switch level

The three types of MOS transistors are primitive description segments: NTRANS, DTRANS, PTRANS. Their 3 interface elements (GATE, DRAIN, SOURCE), of type PIN = VARIABLE (LOGIC3, 'u) are non-directional. They have a STRENGTH attribute (positive integer <100).

The primitive description NODE is the model of an interconnection point. It has a variable number N of non-directional interface PINS. Its 3 other attributes are: SIZE, RISEDELAY, FALLDELAY.

3.7. Imag : addressing the electrical (circuit) level

The primitive description segments are the various dipoles found in an electrical circuit. Their interface is *description component-name (nd FIL P1, P2; in VAR VAL)* where P1 and P2 are two non-directional electrical wires, and VAL is the component value. VAL may be a constant or a function of the circuit state (node voltages, pin voltages and currents). Two functions are attached to each FIL carrier: i (P1) is the current in P1, and v (P1) the voltage. Each primitive component defines an equation on i (P1), i (P2), v (P1), v (P2) and VAL [7.5]. In addition, the compiler automatically generates: *i (P1) + i (P2) = 0* (Kirchhoff laws).

Component type	IMAG identifier	Built-in equation
resistor	R	v(P1)-v(P2)-VAL*i(P1) = 0
conductance	G	(v(P1)-v(P2))*VAL-i(P1) = 0
voltage source	E	v(P1)-v(P2)-VAL = 0
current source	J	i(P1)-VAL = 0
capacity	C	i(P1)-VAL-d/dt (v(P1)-v(P2)) = 0
self inductance	L	v(P1)-VAL*d/dt (i(P1)) = 0

Fig. 7.1. Primitive description segments of IMAG_F.

4. *The cascade hierarchical mixed-mode multikernel simulator (HM3)*

CASCADE allows for mixed level description. Building a simulation model requires algorithms and scheduling modes adapted to each modeling level.

We describe here how CASCADE realizes the scheduling and control mechanisms of the mixed-mode simulation, and solves the cohabitation problems of scheduling modes and different abstraction modeling levels (Circuit, Switch, Gate, Register Transfer and System levels). These cohabitation problems have led us to preserve until simulation run time a hierarchical structure of nested boxes, each of which has its own scheduling mode and its particular modeling level (consequently its particular simulation algorithm).

5. *Scheduling modes*

In a Simulation Model partitioned in "blocks" (a block can be an electrical node, a gate, a set of registers, the operative part of a circuit, etc.), the way of defining the "next event time" and choosing the blocks to be activated, (to simulate the model), is named **scheduling mode**. Various scheduling modes exist in CASCADE according to modeling levels.

5.1. Discrete scheduling mode

The **Time Unit** is abstract and is defined as an Integer. The Time measures the number of clock cycles and the scheduler increases the time by one or more cycles. The model evaluation process at a given time returns the **Next Event Time**. Two kinds of discrete scheduling modes are possible.

5.2. Event driven

The initial circuit is flattened down and forms a network of interconnected elementary blocks. The blocks which have to be activated at a given time (an **Event Time**), are linked to a **Time wheel.** This mode is the one which is used in VHDL or gate level simulation.

5.3. Pre-ordered or levelized

The initial nesting of the circuit is preserved - if possible - by the elaboration program. At each level of the hierarchy a static order is built, taking into account the dependence of variables. If a loop yields this order impossible, then this loop is automatically encapsulated into a new entity created by the elaboration. The scheduler follows this order through the nodes of the hierarchy. It explores the tree in pre-order and, while examining each node, it has to determine if the node is to be activated or not.

5.4. Continuous scheduling mode

This mode is presently used only at **circuit level**: the scheduler, linked with simulation algorithms, chooses the best integration step for the algebro-differential system solver, according to the difficulties encountered and the precision needed (The Time is a real number, measured in seconds).

Two different continuous scheduling modes, (and

simulation algorithms,) are used in CASCADE.

5.5. Global

No network partitioning is done. In this case, the scheduler has only to define the **next event time** which is the sum of current time and integration step. There is only one block which is activated (the analog model block) and the simulator uses its integration scheme to solve the whole system at this time.

5.6. Levelized

The network is partitioned into several analog blocks. At the current time, each block is solved separately, and a global relaxation algorithm stabilizes the whole system. Choosing a levelized vs global continuous simulation mode is left to the designer who knows the electrical model and may exploit some decoupling..

ABSTRACTION LEVEL	SCHEDULING MODE
System level	Control graph Driven
Register Transfer level	Cycle based
Synchronous Gate level	Cycle based
Asynchronous Gate level	Discrete Event Driven
Switch level	Discrete Event Driven
Circuit (with user partitioning)	Continuous Levelized
Circuit (without partitioning)	Continuous global

Fig. 7.2. Relations between scheduling modes and abstraction levels in CASCADE.

6. The Simulation Data Structure

This section defines the first general implementation of mixed mode scheduling, a problem still hot to-day.

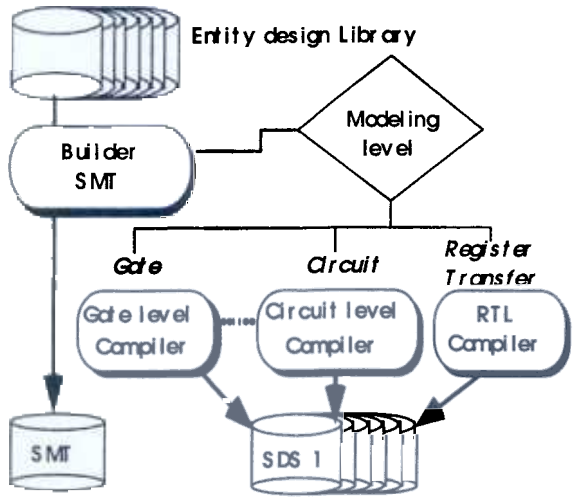

Fig. 7.3. Simulation Data Structures figure

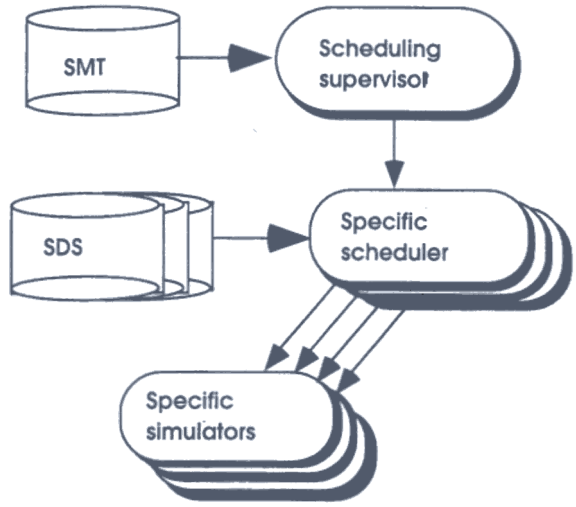

Fig. 7.4. The CASCADE HM3 Simulator.

The CASCADE compiler builds two kinds of data structures to be used by the mixed-mode scheduling: a Scheduling Mode Tree (**SMT**) which allows one to change the scheduling mode in different parts of the same model, and Scheduling Data Structures (**SDS**) adapted to each of the scheduling modes.

For each node of the **SMT**, a specific data structure is associated: the **Scheduling Data Structure** (SDS). This SDS represents the connectivity and, the nesting of the blocks working with the same scheduler.

6.1. The scheduling principle

When the mixed-mode scheduler supervisor finds a **SMT** node, it calls the specific scheduler associated to the mode of this node, and transfers it the two items of timing information required by this scheduler :

* **T1,** the date of the **beginning** of the activity for this block;
* **T2,** the scheduled **end** of activity.

[T1-T2] is a **"Quiet Time Interval"** ***(QTI)***. During this interval, the supervisor guarantees the specific scheduler that no input of the associated block will change.

Then during the evaluation, between T1 and T2, three cases may occur :

1 - A modification is detected on an output pin before time T2 (at T3 where T1< T3<T2). In this case, the control is returned to supervisor with T3 as **Next Event Time.**

2 - No output variation occurs before T2 but the block is not quiescent. In this case, the control is returned to supervisor with T2 as **Next Event Time.**

3 - No output variation occurs before T2 but the block is quiescent. Control is returned to the supervisor with "NIL" as **Next Event Time** (latency information).

7. Conclusion

A complete prototype of the HM3 simulator was implemented on VAX/VMS by the end of 1986. The prototype was tested on a large microprocessor model, which embedded components described at **all** CASCADE levels. The emergence of the VHDL standard prevented us from completing an industrial development out of this successful university project, and going to the marketplace. Yet the experience gained in CASCADE was a basis which contributed to the VHDL/AMS standardization effort and will continue to contribute to VHDL extensions. No commercial tool, 12 years later incorporates all the similation capabilities demonstrated on the CASCADE prototype. The OMI /IEEE standard recently voted proposes a scheduling of OMI models which follows the principles described above.

CASCADE References

[7.1] D. Borrione: LASCAR: a Language for Simulation of Computer Architecture, Proc. CHDL'75, New York, USA, Sept. 1975

[7.2] D. Borrione, J.F. Grabowiecki: Informal introduction to LASSO: a Language for Asynchronous System Specification and simulation, Proc. Euro-IFIP 79, London, Sept.79.

[7.3] D. Borrione, R. Piloty: The CONLAN Project: concepts, implementations and applications, IEEE Computer Magazine, Vol 18, N°2, Feb. 1985

[7.4] C. Le Faou: Un programme général de simulation de circuits electroniques: IMAG, Electronique et Micro-Electronique Industrielle, Octobre 1974.

[7.5] C.W. Rose: "LOGOS and the software engineer", Proc. Fall Joint Computer Conf. 1972 pp. 311-323

[7.6] J. Mermet: Etude methodologique de la conception assistee par ordinateur des systemes logiques, Thèse d'Etat es-sciences mathematiques, Universite de Grenoble, France, 21mars1973.

[7.7] J. Mermet: Circuits and System Computer Aided Design and Engineering:CASCADE, Proc. CAPE83: 1st Int. Conf. on Computer Applications in Production and Engineering, Amsterdam, The Netherland, April 1983, pp. 245-262.

[7.8] R. Piloty, M. Barbacci, D. Borrione, D. Dietmeyer, F. Hill and P. Skelly: CONLAN Report, Lecture notes in Computer Science 151, Springer-Verlag, Berlin, 1983.

VIII. Comparison of the Selected Languages

The sample of languages presented above provides a large coverage of the concepts and features of modern HDLs among which we have chosen the following fifteen characteristics to qualify more precisely these languages

-1- Combination of programming-language syntax with hardware semantics:

European originated languages such as PASCAL, ADA or MODULA have contributed to works such as MIMOLA, ART, DACAPO or even VHDL. But also international de facto standards (with strong European roots) like ALGOL or CONLAN influenced CASSANDRE, ELLA and CASCADE.

-2- Formal Semantic Models :

Exist in DACAPO, ELLA, or CASCADE., using PETRI-nets, process algebra or Base-CONLAN.

Dont exist in VHDL or VERILOG

-3- Hierarchy & Modularity of design descriptions.

-4- Generics:

-5- Unconstrained blend of behavior with structure in a design description:

-6- Abstract data types, private types

(exist in CASCADE, DACAPO, ADA but neither in VHDL nor in VERILOG.)

Concept ->	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CASSANDRE	ALGOL	NA	UNIT (FIRST MODU-LAR HDL)	Y	Y	NA	MULTI CLOCK TD	MDO	SV	NA	NA	NA	NA	NA	FSM HIE-RAR-CHY
MIMOLA	PASCAL	NA	NA	NA	NA	NA	SINGLE CLOCK	BIT VEC-TOR	NA	NA	NA	NA	Y	NA	ROM
ART	PASCAL	NA	UNIT	Y	Y	NA	MULTI CLOCK	MDO	SV	NA	NA	NA	NA	WDL	FSM
DACAPO	MODULA	PETRI NETS CON-LAN	MO-DULE	Y	Y	Y	DYNA-MIC TD INTER-RUPT HAND.	MDO	Y	Y	NA	NA	NA	TGE ESN TER	CTRL GRAPH MES-SAGE PASSING
ELLA	ALGOL	PRO-CESS ALGE-BRA	FUNC-TION	Y	Y	Y	AUTO CLOC-KING	MDO	LEFT TO USER	Y	NA	Y	Y	NA	FSM
CASCADE	CONLAN	BASE CON-LAN	DES-CRIP-TION	Y	Y	Y	DYNA-MIC TD TPDL	MDO	Y	Y	Y	Y	Y	NA	CTRL GRAPH FSM HIERAR-CHY

Fig. 8.1. Tabulation of 15 Language Properties Versus The Languages Examined. The numbers are indexed to the language properties listed. Abbreviations: NA : Not Applicable, TD : Transport Delay, MDO : Multi -Dimensional Objects, SV: Sub-variable, WDL: Waveform Description Language, TPDL: Temporal Profiles Description Language

IX. More Recent Work

1. VHDL93:

VHDL87 cannot be claimed an European language. But it is inspired by ADA which was one.

After declassification by the DoD the influence of European partners on the evolution of the standard became strong, and this is reflected in VHDL93. Europe was also the earlier adopter of VHDL which is still the dominant language there today. This can be qualified a cultural problem. Europe has always been a region of proliferation of languages (whether natural or formal). There was a strong need for standard. VHDL did not bring unfamilair concepts and had a flavor of neutrality: it was immediately adopted.

Then we cannot forget VHDL in this Europe-centric paper. VHDL brings a few features new to previous languages:

- Service modularity with **packages** and **libraries.**
- **Generalized modularity of entities,** with the notion of architecture independent of an entity. A given piece of design, the entity can have several views in VHDL, instantiated as « architectures ». Then VHDL encompasses several levels of abstractions, within the limits of « discrete event driven timing ».
- **Functions** and **Procedures** (like CONLAN)
- **Overloading** of operators by functions
- Description of **truth tables**
- Process network execution model which provides concurrency and initializes the model at simulation.

2. VHDL-AMS.

Among European requirements for VHDL93, there was a strong demand for mixed analog/digital modeling and simulation. The timeframe appeared, during the VUFE in Santander, too short to introduce this evolution into VHDL 93. Then a new IEEE working group was created, which has been chaired by European people until now. VHDL/AMS is a true superset of VHDL which brings new concepts such as **nature** which relates to their specific domain of application (electrical, mechanical, ..) the variables of the analog model. When adding a continuous time model semantics to the event driven model of computation VHDL-AMS does not bring a new concept as this was previously in some mixed mode languages (the IMAG level of CASCADE for example), but this is for the first time done at all levels of the VHDL description, using for example the **across** and **through** new statements, not only in distinct entities VHDL-AMS can be used also at **system level,** to describe models and their environment with a continuous/discrete timing scheme.

3. Objective VHDL.

Objective-VHDL is an object-oriented extension of VHDL developed within an European R&D ESPRIT project called REQUEST, mainly at university of Oldenburg. The concepts introduced by objective VHDL are **entity classes, architecture classes,** and **type classes** The benefit of such an object-oriented extension depends on the availability of class libraries (for example to store and re-use a communication channel or protocol after it has been implemented once). Objective VHDL is a true superset of VHDL with only two new keywords: **class, abstract.**

A complete prototype of the Objective VHDL to VHDL translator was due on summer 1998.

4. VHDL+

This VHDL extension was developed at ICL (UK) as an attempt to address some system level requirements not fulfilled by VHDL. The initial work was a language and set of tools called CHISLE (for Combined Hardware and Interface Specification Language for Engineers). The syntax was inspired by Algol68. Then a fast VHDL87 simulator was developed, followed by VHDL+, when it appeared that most EDA tools would be based on this standard.

The new concept brought by VHDL+ is that of **Interface primary unit,** which defines the communication between entities, can be re-used, and is multi-level. In the interface the communication protocol can be defined at different levels, so that different, more or less refined, views of 2 entities can always communicate through the same interface unit. This provides a kind of polymorphism to linking entities to each other at different abstraction levels that VHDL does not have. Interfaces contain: **transactions** (2 ways exchange), **messages, packets,** signals and **compositions** (mapping from one level to another), and some constructs, serial blocks, parallel blocks, clock, after, pause, aimed at describing more easily non event-driven temporal properties at high level. A new kind of port, the **interface port,** is added to the VHDL entity interface to be the link with Interface units. The architecture contains other additions: interface signals, interface map, activities, and **send** and **receive** to allow the use of interface items from processes and activities.

X. Concluding Remarks

It is clear that a modern language like VHDL, apart from being THE recognized standard in Europe and offering many commercial quality environments, does not yet incorporate many of the features implemented in some of the ancestor languages cited above. Registers, Clocks, FSMs, communication and synchronization mechanisms, object constructs, private data types, all these proven concepts are requirements for **VHDL 200X,** the version of the standard that will follow **VHDL 98,** whose preparation has started already. Are also lacking continuous or system level modeling capabilities. The continuous modeling capability hopefully will come with **VHDL-AMS,** but system level description and modeling, which is the main concern of European system houses, will not be provided by VHDL extensions only. VHDL is already a complex language and there is a limit to the increase of

complexity that users can further accept and EDA tool providers implement.

Design environments will have probably to accommodate a variety of different languages at a level ABOVE VHDL, but DEEPLY INTERFACED with it. The above mentioned extensions will shorten the distance to these other languages and certainly facilitate the establishment of a seamless design flow, from system specification to circuit level, in a few years.

The **future of HDLs is not an HDL.** HDLs have made it possible to design multi-million gate chips and to follow the Moores law over a third decade. But a paradigm shift has occurred: what is to be designed **now** is System on Chip (SOC). SoC is no-longer hardware only, but more and more embedded software. The hardware itself is no longer digital circuits only but also analog, RF, sensors, microwaves, batteries, micro-machine, optical devices.....why not living cells?

If we only restrict ourselves to hardware/software co-design and verification there is today a Babel Tower situation among dozens of languages and formalisms, which recalls us the situation of HDLs around 1975, when the CHDL conference decided to create a unification group called CONsensus LANguage.

We need another CONLAN effort today at system level. But this effort can last several years. In the meantime many of these languages will continue to be used beneficially in different application domains. And the design environments will have to be multi-language for an undefined period of time.

To accomodate all these languages in a system design space, an **architecture specification language** or notation is still to be defined, which will provide bridging semantics between different languages and computation interpretation mechanisms, between different time models. Like in CONLAN, the goal would not be to define Yet Another Language, but to define the underlying semantics and provide the basic mechanisms to re-build the candidate system level languages on it.

Some projects exist, in Europe, around Esterel for example, with EC an extension of C with synchronisation mechanisms translatable into Esterel (Lavagno). Already mentioned also is the project of the CADENCE lab in France to insert Esterel in their Polis environment. Similarly the VERILOG team in Grenoble works on integrating LUSTRE in their Object-Geode environment. Many of these multi-language environments exist now in large European companies designing systems. As an example, at Italtel, OCCAM, Objective-VHDL and C++ are used in conjunction with retargetable code generators to implement DSPs and embedded systems. But many steps in the design flow are not yet automated, and the verification tools available dont reach the necessary (and fast growing) level of performances.

Many interesting works, mainly in Europe and USA, have tried, over the last 30 years, to give a sound theoretical foundation to the construction of large software systems. The domain of HDLs or SLDLs does not require different theories. It is clear that the advanced software concepts have been adapted after a certain delay to the specification of hardware. The tremendous speed of micro-electronics technology progress may have hidden this trend for a while: many difficult problems have not been solved, but just erased by the brute force of a doubling of performances every 18 months. But at the time of System-Chip design the unsolved problems re-appear and a look back to mathematical background imposes itself.

Following the pioneers, Dijkstra, Hoare,..., J. R. Abrial has made an impressive and clear synthesis of the knowledge in this domain with B. **B** is a method for specifying, designing, and coding software systems. The basic mechanism of this approach is that of **abstract machine,** a concept close to modules, classes or abstract data types. The task of B is to accompany the technical process of program construction by a similar process of proof construction, which guarantees that the proposed program agrees with its intended meaning. System on Chip specification and design demand a similar approach. The solution to the challenges of system level modeling will be met, only as Abrial says, "by *returning to mathematics*".

General Bibliography

[G. 1] J.R. Abrial, *The B-book . Assigning programs to meanings*, Cambridge university press, 1996.

[G. 2] M. Barbacci, D. Borrione, D. Dietmeyer, F. Hill, R. Piloty, and P. Skelly, *CONLAN Report*, Lecture notes in Computer Science N° 151, Springer Verlag, 1983.

[G. 3] M. Barbacci, T. Uehara, guest editors, IEEE Computer (special issue on hardware description languages,) Vol. 18, No 12, February 1985.

[G. 4] D. Borrione, JF Grabowiecki, "Introduction to «LASSO» Language for Asynchronous System Specification and simulation," *Proc, EuroIFIP 79*, London, sept 1979.

[G. 5] D. Borrione, "CASCADE," in *Fundamentals and Standards in Hardware Description Languages*, KLUWER ACADEMIC, 1993.

[G. 6] R.T. Boute, "On the shortcoming of the axiomatic approach as presently used in Computer Science," *Proc IEEE COMPEURO 88 - Systems design: concepts, methods and tools*, pp. 184-193, April 1988.

[G. 7] R.T. Boute, "Fundamentals of Hardware Description Languages and Declarative Languages," in *Funda-*

mentals and Standards in Hardware Description Languages, KLUWER ACADEMIC, 1993.

[G.8] L.A. Cherkasova, V.E. Kotov, "Structured Nets," *Proc. MFCS81*, Springer LNCS 118, 1981.

[G.9] Y. Chu: An algol-like Computer Design Language, *Comm. of ACM*, October 1965, pp 607-615.

[G.10] T.D Friedman, "ALERT: a program to produce logic design from preliminary machine description ," RC 1578, March 24, 1966, IBM Research.

[G.11] D. Harel, "Statecharts: A visual formalism for com-plex systems," *Science of Computer Programming*, 8, 1987.

[G.12] R. Hartenstein and P. Liell, *KARL-2 Language Reference Manual*, Kaiserslautern 1983.

[G.13] A.D Falkoff, K.E. Iverson, and E.H. Sussenguth, Formal description of system/360 *IBM sys. J.*, vol. 3, pp 198-262, 1964.

[G.14] A. Jerraya, H.Ding, P. Kission, and M. Rahmouni, *Behavioral synthesis and component reuse with VH DL*, KLUWER ACADEMIC, 1997.

[G.15] P. Liddell, "Decoupage syntaxique de systemes logiques decrits en CASSANDRE," *These de 3ieme cycle*, Grenoble, March 1970.

[G.16] B. Lutter, W. Glunz, and F.J. Rammig, "Using VHDL for Simulation of SDL Specifications," *Proc. Euro-DAC92*, 1992.

[G.17] P. Marwedel, "A new synthesis algorithm for the MIMOLA software system," *23rd Design Automation Conf.*, pages 271277, 1986.

[G.18] J.D. Morison and C. Newton, "ELLA, a Language for the Design of Digital Systems," in *Fundamentals and Standards in Hardware Description Languages*, KLUWER ACADEMIC, 1993.

[G.19] J.D. Morison and A.S. Clarke, *ELLA 2000, A Language for Electronic System design*, McGrawhill, October 1993.

[G.20] R. Piloty, "REGLAN," in *Fundamentals and Standards in Hardware Description Languages*, KLUWER ACADEMIC. 1993.

[G.21] F. Rammig, "DIGITEST II: An Integrated Structural and Behavioural Language," *Proc. IFIP CHDL75*, 1975.

[G.22] F. Rammig, "The Hardware Description Language DACAPO III," in *Fundamentals and Standards in Hardware Description Languages*, KLUWER ACADEMIC, 1993.

[G.23] F. Rammig, "System Level Design," in *Fundamentals and Standards in Hardware Description Languages*, KLUWER ACADEMIC, 1993.

[G.24] H.P Schlaeppi, A formal language describing machine logic, timing and sequencing (LOTIS), *IEEE Trans. on Electronic Computers*, vol EC-13 pp 439-448, Aug. 1964.

[G.25] B. Hennion, G. Mazare, P. Senn, and H. Tahawy, "New Implementation Technique for the Simulation of Mixed (Digital-Analog) VLSI Circuits," *Proceedings from the international conference on Computer-Aided Design*, Santa-Clara, CA, Nov. 1987, pp 396-399.

[G.26] A. Vachoux, "Analog and Mixed-Level Simulation with Implications to VHDL," in *Fundamentals and Standards in Hardware Description Languages*, KLUWER ACADEMIC, 1993.

[G.27] E. Villar, L. Berrojo, and P. Sanchez, "High-level synthesis and simulation with VHDL," *Proc. of the Second European Conference on VHDL*, Sockholm, Sept. 8-11, 1991, pp 62-69.

[G.28] *VHDL Modeling Terminology and Taxonomy* , RASSP doc, Sept 9, 1996.

[G.29] G. Zimmermann, "The MIMOLA design system: A computer aided digital processor design method," *Proceedings of the 16th Design Automation Conference*, pages 5358, 1979.

[G.30] Y. Chu et al., "Three decades of HDLs: Part 1, CDL Through TI-HDL," *IEEE Design & Test of Computers*, June 1992.

Dr. Jean.P.Mermet Directeur de recherches at the CNRS (TIMA lab., Grenoble University) and director of the European CAD Standardisation Initiative (ECSI). J.M holds an engineering degree (ENSIMAG 66), a master in Mathematics (1967), and a DESS degree in Economics (1977). He achieved a 'Doctorate thesis in Informatics (1970) and a Docteur dEtat thesis in mathematics (1973). He has been working on Hardware Description Languages since 1966, mostly at the University of Grenoble. The languages and associate simulators, CASSANDRE, LASCAR, LASSO, IMAG2, CASCADE, were developed by him or in his team. Besides this research activity JM has also held the positions of : Delegue aux Relations Industrielles en region Alpes (1974-78), Directeur Scientifique of the Mediterranean Institute of Technology / IMT (1988-92). JM has created: - the Association MICADO (1974) for the promotion of CAD in all domains. - The European event MICAD (Paris, 1980), - The conference EuroVHDL (Marseilles, 1990), - The conference APChDL (Brisbane, 1993), - The European association ECSI (Grenoble, 1993). JM has been the director of 4 NATO advanced Study Institutes (1971, 93, 96, 98). He has a long experience of European projects as he was the leader of the first of them in micro-electronics: the CERES project (feb 1983). JM has edited 7 books and written over 100 publications. He is editor in chief of « the ECSI letter » and the « VHDL newsletter ». He is presently the chair of the IFIP 10.5 Working Group. JM has been involved for a long time in the standardisation activities: the CONLAN project, the VHDL Analysis and Standardisation Group (IEEE), which he co-chairs from the creation , the Design Automation Standardisation Committee (IEEE-DASC). In 1995, he received the « meritorious service award » from the IEEE, for these achievements.

Peter Marwedel received his Ph.D. in Physics from the University of Kiel (Germany) in 1974. He worked at the Computer Science Department of that University from 1974 until 1989. In 1987, he received the Dr. habil. degree (a degree required for becoming a professor) for his work on high-level synthesis and retargetable code generation based on the hardware description language MIMOLA. Since 1989 he is a professor at the Computer Science Department of the University of Dortmund (Germany). His research areas include hardware/software codesign, high-level test generation, high-level synthesis and code generation for embedded processors. He is one of the editors of the book "Code Generation for Embedded Processors" published by Kluwer Academic publishers in 1995.

Dr. Rammig studied mathematics and informatics at Bonn University, Germany. He obtained a Ph.D. title in informatics from Dortmund University, Germany in 1977. Since 1983 he is Professor for Practical Informatics at Paderborn University, Germany. Dr. Rammig is one the two directors of C-LAB, the joint R&D lab of Paderborn University and Siemens AG. At the same time he has one of the chairs of Heinz Nixdorf Institute of Paderborn University, an interdisciplinary institute between informatics and engineering. He is Vice Chair of the German Informatics Society (GI) and represents Germany in TC10 (Computer Technology) of IFIP. After working in the area of hardware design, especially in the area of hardware description languages his current research interests are focused on distributed embedded real time systems.

Dr. Cleland Newton is Principal Scientist in the System Assurance Group at the Defence Evaluation and Research Agency (DERA) at Malvern, UK. He has been involved in the design of VLSI circuits since the early days of Hardware Description Languages. He was one of the pioneer users and contributors to the ELLA HDL at RSRE (which is now DERA Malvern). He has worked on correct-by-construction methods for hardware using ELLA, VHDL and VDM. His current interests are the safety, supportability and reliabilty of electronic hardware. This includes the use of formal methods to achieve design assurance for safety critical applications and the investigation of prognostic techniques to obtain higher reliability for complex electronic systems.

Dominique Borrione is Professor of Computer Science at Université Joseph Fourier, Grenoble, France, and leads the research group on "Modeling and Verification of Digital Systems" at TIMA Laboratory. Previously, she had been a Research Scientist at CNRS, then a Professor at the University of Marseille until 1988. She received a PhD in 1976 and a Doctorat d'Etat in Computer Science in 1981, both from the Univesity of Grenoble. Her research interests include

Hardware Description Languages, CAD for large digital electronic systems, and the application of formal methods in design and verification. She is the author of numerous publications on these topics, and has served in many conference and workshop committees. She was a member of the Conlan Working Group, and is active in various aspects of VHDL standardization. She is a member of the IFIP Working Group 10.5.

Claude Lefaou (born on May 20th 1938) received his diploma of Electrical Engineering from the INPG Grenoble in 1962. His fields of interest are CAD, Simulation, Hardware Description Languages, and he is author or co-author of about 20 publications in these domains. He is author of IMAG2 (1970) and IMAG3 (1972), simulation programs at the circuit level and he was technical manager of the CASCADE project (1981-87): CAD system for VLSI circuits and systems. From 1990 to 1995 he was Adjoin Director of ARTEMIS Laboratory. He is currently "Ingenieur de Recherche" at TIMA (CNRS laboratory) and he is working in the field of Formal Verification from VHDL descriptions.

Specification and Synthesis of Exception Handling in Grammar-based Hardware Synthesis

Johnny Öerg, Anshul Kumar, Ahmed Hemani, and Shashi Kumar

Abstract

ProGram is a grammar based specification language aimed for specification of protocols, interfaces and control dominated functionality. Specification of exception handling functionality and its automatic implementation are key requirements for a robust design methodology. We have extended ProGram to include specification of a wide range of exception handling functionality: reset, to deal with exception handling needs of global nature; interrupt, for representing exception handling on a hierarchical basis; and error sequences, to handle situations when inputs are not according to the specified grammar. We have also enhanced our synthesis algorithms and relieved them from some earlier limitations, to allow the ProGram compiler to deal with the exception handling extensions to the ProGram language. The tool based on these techniques has been used for the implementation of 1) a send/receive protocol supplied by industry, 2) a bus-arbiter and 3) the Intel 8251A Personal Communication Interface.

Keywords : Exception Handling, ProGram, Specification, Synthesis

I. Introduction

Exception handling is necessary to be able to describe hardware and real-time systems in an efficient way and is a key aspect of any system functionality, where it takes the form of reset sequences, interrupt service routines and error handling. The key characteristic of exception handling is its global nature. For instance, a reset would bring the state machine to a known state from any state and clear the storage elements. Reset, though a common hardware construct, does not have any explicit support in VHDL, in which any exception of global nature is treated as any other condition. This makes it very cumbersome to specify exception handling functionality. Other languages, like SpecChart[1][2], have explicit support for specification of exception handling functionality. Interrupts are of a more local nature than a reset, and specify how the protocol should react if a high-priority situation occurs, caused by some hardware failure, some illegal operation performed elsewhere in the system, an interrupt request or some other external event. Error handling is crucial since it makes it possible to detect and recover from error situations.

ProGram [10][11][12] is a grammar based specification language aimed for specification of protocols, interfaces and control dominated functionality. Synthesis from grammar-based descriptions result in efficient hardware. In addition, the ProGram methodology allows to perform design space exploration of the port-sizes by letting the designer pose the input and output port-sizes as constraints to the tool. The input port constraints are used to partition the message specified by the grammar rules into chunks of appropriate size. The output port constraints are used in the same way to partition the output assignments into chunks of appropriate size. The partitioned output assignments are then scheduled over the available stages of the input message.

ProGram has been extended to include exception handling [13]. The inclusion of exception handling constructs results in constraints which make the previous output scheduling algorithms [12] produce inefficient or, in some cases, wrong results. The synthesis algorithms have been extended to handle the new exception handling constructs [14], but the

Johnny Öerg and Ahmed Hemani are with the Electronic Systems Design Laboratory, Royal Institute of Technology, ESDLab/KTH-Electrum, Electrum 229, S-164 40 Kista, Sweden.

Anshul Kumar and Shashi Kumar are with the Department of Computer Science & Engineering, Indian Institute of Technology, New Delhi, India.