Using Ada as a Language for a CAD Tool Development: Lessons and Experiences

Jamal Guennouni

Signal Department
ENST CNRS UA-820
46, Rue Barrault
75634 Paris Cedex 13, France.

1. INTRODUCTION AND OUTLINE OF THE PAPER

Since the advent of Ada, many papers have been published which deal with Ada's pros and contros. Few of them present comparative advantages, drawbacks and practical experiences based on a real development effort. The aim of this paper is to report on our conclusions using Ada in a complex development project which concerns both the development of a CAD tool and the simulation of a VLSI circuit dedicated to speech analysis-synthesis. The CAD tool consists in a timing analysis program which makes it possible to schedule hardware component models according to their activation times. Specification and verification of timing constraints such as propagation delays are thus possible. Therefore, the CAD tool may be referred to as, either the scheduling tool, or the timing analysis tool and acts as a simulation environment. Ada is both a hardware description language and centralized simulation environment since the hardware components have been entirely described in Ada as well as the simulation environment (i.e., scheduling tool). Our approch is thus uniform, unlike the ones followed by other authors (e.g., Hill [12]) where the hardware description language is ADLIB while the simulation environment is PASCAL. An overview of our CAD tool can be found in [11]. Accordingly, our discussion will focus on the use of Ada in the hardware simulation field. The timing constraints which can be expressed thanks to the CAD tool constitute a timing model; the latter can "stick" to a purely functional description to perform the logical and temporal behaviors of a set of components over a given period (i.e., specified by a designer). In particular, timing errors which are common in VLSI design can be detected and eliminated.

The remainder of this paper is organized as follows. In section 2, we present the goals of our study along with the application case chosen here. In section 3, we explain the reasons why we have chosen Ada. The main aspects of the Ada language used in our experience are illustrated in section 4 using a few examples. Consequently, these examples will show us the real contribution of Ada. Section 5 deals with our experience with the use of Ada for tasking. Some of the problems faced will be outlined and the differences compared to some other languages will be mentioned. In section 6, comparisons between our present experience and previous Ada related work in the hardware description field will be made. Section 7 gives some simulation experiments. Lastly, conclusions and final recommendations based on our experience will be stated in section 8.

2. CASE STUDY

The framework of our study is the design of a VLSI circuit dedicated to speech analysis-synthesis. Therefore, this circuit belongs to the ASIC (Application Specific Integrated Circuits) family. The need for this circuit family is continually increasing (*). The main potential advantage of the ASIC over the universal (i.e., not specific) circuits is a better performance-cost ratio. The production of ASIC is usually lower than universal circuits's. In order to reduce the total cost (i.e., production plus design costs) associated with ASIC, it is imperative to decrease their design cost. To achieve this goal, a designer must have not only a set of efficient CAD tools and coherent design methodology but also appropriate design and implementation languages. As we shall see in this paper, the impact of the implementation languages on software cost, productivity and quality is important especially when it comes to hardware descriptions. Indeed, simulation is a common tool for analyzing circuits prior to prototyping. Software representations of hardware design which usually consists in several thousands lines of code must thus be built in an appropriate language.

For the clarity of this paper, we will focus our discussion on the analysis part of the speech analysis-synthesis circuit and more particularly on some of its hardware components. The aim of the analysis part is to provide the synthetic part with the information necessary to generate the synthetic speech.
The principle of the analysis phase is the following. A model representing the real mechanism responsible for the sound generation (i.e., vocal tract) is defined at first. The most widely used model is a linear filter whose spectral and parametric characteristics are assumed to change only slightly over short observation ranges (i.e., over signal frames varying from 10 to 40 milliseconds). This filter, in

(*) we expect that the European ASIC market should reach 2 billions dollars in 1990, i.e., 20% of the whole integrated circuits market

turn, is represented by a mathematical model. The output of the filter is obtained by applying the principle of the Linear Prediction Coding (LPC) which states that one can approximatively predict the actual speech sample as a linear combination of the p past samples (e.g., see Markel and Gray [14]):

$$(s_{predicted})_n = -\sum_{i=1}^{p} a_i * s_{n-i} \quad (1)$$

where $(s_{predicted})_n$ is the predicted speech sample, p the order of the predictor (typically between 8 and 16) and a_i the predictor coefficients.

To measure the model's accuracy (i.e., its ability to faithfully represent the real mechanism responsible for sounds generation), we compare the speech signal to be analyzed (i.e., original or reference signal) with the filter output $(s_{predicted})_n$ given by the above equation. We therefore consider the following difference given by:

$$\varepsilon_n = s_n - (s_{predicted})_n \quad (2)$$

where ε_n is called the predictor (or residual) error and represents the difference between the actual value and the predicted value.

To determine the model parameters a_i, we minimize a squared criterion (least squared criterion of the prediction error):

$$\theta_n = \sum_n (\varepsilon_n)^2 \quad (3)$$

where θ_n is the mean-squared error.

The choice of the summation range depends on the method selected (Markel and Gray [14]): auto-correlation method; covariance method. The main advantage of the LPC method over other coding techniques lies in the fact that it represents a good compromise between the model's accuracy (its ability to represent the real phenomenon) and simplicity (small number of parameters). Practical considerations have led us to slightly modify the series of calculations given by the classical auto-correlation method. Indeed, we have decided to calculate a sequence of p coefficients called PARCOR (for PARtial CORrelation) instead of the p predictor parameters a_i. The main advantage of our choice is that the PARCOR coefficients are well suited for VLSI and microprocessors implementations (e.g., their dynamic can be known with accuracy and the calculating algorithm is robust). Choices are made in order to select those algorithms whose structures are more amenable to VLSI implementations.

Figure 1 shows the different phases of the LPC analysis based on the choices we have made. In fact, the complete process of the LPC analysis consists in :

- the pre-emphasis of the input speech signal which is sampled beforehand.

- the appropriate windowing of this signal (e.g., use of a Hamming window).

- the correlation computation of the speech samples.

- the PARCOR coefficients calculation.

- the inverse filtering of the input speech signal to obtain residual error.

The analysis part of our circuit is therefore made up of three main functional blocks: correlator, PARCOR extractor and lattice filter. Each block has its own sequencer driven by a global sequencer.

3. WHY ADA ?

Roughly speaking, our study has consisted in two main phases. The first one may be referred to as the software phase. Its target is double :

- to determine the algorithms class which will serve to perform the calculations (e.g., LPC analysis).

- to test these algorithms on the basis of a set of selected criteria such as robustness and sensitivity of the speech synthetic quality to the data accuracy.

The second phase really concerns the hardware design. In this phase, we describe the hardware which performs the calculations specified by the algorithms selected in the first phase. A hardware specification consists of a list of hardware components along with the description of their interconnection and of a controller that will sequence the flow of data through the hardware network.
For both phases, a specification and simulation language is needed. Two main alternatives are possible: either we choose a general purpose programming language or a specific simulation language.

To opt for a general purpose programming language is useful especially for the first phase of our study (i.e., software phase). As for the hardware phase, one can think of extending a given general purpose programming language to provide it with some simulation facilities as has been formerly done with other languages such as Pascal (e.g., ADLIB is a superset of Pascal: see Hill [12]). Nevertheless, the extension of a language has some main drawbacks. It requires the development of a new compiler along with its associated debugging support environment. Besides, the relatively important investment required by such a task can divert people from their real problem (e.g., design of a VLSI circuit in our case) or at least put them out of touch with it when this development effort is carried out by a specialized team. The question arises even less as whether we extend Ada or not: Ada has been standardized and the DoD (Department of Defense) is reluctant to any extension.

Despite the fact that in theory there is no reason why we should prefer a language to another, practical considerations prevail. In our case, we have decided to use Ada throughout our work (i.e., software and hardware phases) for the reasons mentioned below. Obviously, only the main Ada advantages will be given because of the limited length of this paper.

At first, with regard to the potential advantages of Ada compared with other general purpose programming languages (e.g., Fortran or Pascal), Ada has some sophisticated structures such as packages and separate compilation facilities which enable modularity and hierarchization useful

to develop large software models. These facilties are therefore especially interesting in our case since the description of a VLSI circuit like ours contains many thousands of lines of code. Moreover, the tasking model is provided in Ada. Within the hardware description framework, this facility is appealing since tasks allow one to elegantly model a set of hardware elements whose activities evolve in parallel. Nevertheless, we will outline in section 5 some problems we have come accross with when using Ada task-based models. Furthermore as we shall see Ada generics make it possible to develop flexible software components. Advantages of the generics concept within the the VLSI design framework will be explained in next section.

The other main alternative which we have mentioned is to select a specific simulation language. Here again, several choices are possible. Indeed, at a high level of abstraction (i.e., behavioral level), our circuit can be viewed as a synchronous system driven by a global (i.e., master) clock. One can therefore think of using either a synchronous programming language (e.g., ESTEREL [4], LUSTRE [5]) or a discrete simulation language (e.g., SIMULA [7], GPSS [21]). Indeed, both kinds of languages are appropriate for synchronous system simulation. As for the hardware phase, one can think of using a hardware description language. In fact, there is no use of a specialized simulation language in our experience because of the reasons explained below.

As for the discrete event simulation languages, their deficiencies are now well recognized: a discrete simulation language on the one hand often implements only one technique among the three main ones and on the other hand these languages suffer from poor control structures and portability. Furthermore, Ada makes it possible to achieve modularity and hierarchy to tackle complex problems such as the description of a VLSI circuit.

The drawbacks of the synchronous programming languages are mainly due to the fact that they are relatively new. Therefore, they are not very operational yet and some deficiencies have to be overcome (e.g., lack of an efficient debugging support environment and separate compilation facilities).

Finally, even though Ada was not designed with hardware in mind, there are several features in Ada which are appealing for hardware modeling and which will be shown by using a few examples in next section. However, we can already mention some of the advantages found in Ada:

- Ada fits different level of descriptions (e.g., behavioral, logical and switch levels), unlike other Hardware Description Languages (HDL's) which are limited by a particular level of abstraction (e.g., the LSL-LOCAL language in the LAMP system [6] is dedicated mainly to the gate level).

- Ada has a better portability than most commonly used HDL and it is standardized.

- Ada is not specific either to a particular type of architectures or to a particular level of simulation.

The advantages above are based on a comparison with "conventional" HDL. But, for some years, languages such as Prolog and Lisp have been used for hardware simulation. Thus Batali [3] at the MIT University has used Lisp to describe a circuit at various levels and notably for the functional specification. Mouly [16] gives an example of the use of Prolog to resolve a routing problem.

Despite the fact that Artificial Intelligence (AI) languages such as Prolog and Lisp are interesting to use in hardware descriptions and simulation (e.g., to select a best solution if some already exist) their drawbacks cannot be overlooked. According to Mouly's paper, for instance, Prolog is too slow (e.g., it cannot be compiled efficiently) and several attempts have to be made before a solution can be found even when faced with a simple problem. When several solutions can be found, these arise mainly from the backtracking feature of Prolog. To reduce the number of possible solutions, rules must be introduced. As these rules depend on the application case, one cannot give a general method to specify them.

4. SOFTWARE TECHNIQUES IN ADA FOR HIGH LEVEL HARDWARE DESCRIPTIONS

The aim of this section is to present comparative advantages and disadvantages of Ada based on our practical experience. For the clarity of this discussion, we will focus on the main characteristics of Ada which have been very helpful in our development effort. The examples selected to illustrate these characteristics will be taken in the hardware description field (i.e., in the second phase of our work). We will take, on purpose, simple examples to avoid the unnecessary details relevant to the design of some complex digital systems.

4.1. Hierarchical Descriptions

The goal is to build high level complex hardware components from lower level elements in a hierarchical manner. However, this cannot always be done since some languages do not provide any facilities to use low level hardware models to build more complex ones. Therefore, a programming language which does enable hierarchy is appealing. In this respect, Ada reveals to have been a good choice since Ada packages allow high level hardware models to be built from lower models. For example, the correlator block shown in figure 1 is composed of a bit serial multiplier/accumulator and of a set of registers and multiplexers.

The principle of our implementation methodology is the following. The behavior of a given hardware component is described as a subprogram (in the Ada sense) encapsulated in a package. We therefore have, whatever the level, a package which manages a particular kind of hardware component. For instance, we have a package for the accumulator (referred to as the bit_serial_adder in figure 2), another one for the multiplier and so on. In order to build a higher element (e.g., correlator) we use the services provided by the packages implementing the low level components (e.g., multiplier and accumulator). Moreover, to design complex elements, we begin by designing, implementing and testing the components of lower level (subcomponents).

The notion of packages combined with the separate compilation facilities make it possible to reduce the sensitivity to changes of software representations of hardware

components. Indeed, it is possible to replace an algorithm which implements the behavior of a given hardware component by another one as long as we keep the interface unchanged. This interface corresponds to the package specification part which models this hardware component. On the other hand, as the behavior of a same component can be implemented in different manners, circuit design and consequently, circuit simulation, is a repetitive process: a designer has to reconsider the choices he had made. In particular, the implementation choices already made have to be revised. Thus it is desirable that the programming language selected should enable a switch between a given algorithm implementation and any other provided that the interface does not change. With Ada, the hierarchization and the reduction of sensitivity of programs to changes are possible thanks to the packages notion and separate compilation facilities.

4.2. Generic Models

One of the main advantages of software over hardware is that it is more supple: it is more difficult to change hardware components or connections than to change some lines of code corresponding to software representations of these hardware components. However, this flexibility inherent in software is not sufficient once one knows that complex models such as the ones associated with VLSI circuits may consist in several thousands lines of code. This flexibility must therefore be enhanced. One of the most interesting methods in this respect is the one which consists in parametring programs. In our case, this mainly implies that the code does not change but the internal or/and external data handled by the software representations can have their formats or/and values changed. Ada thanks to the generics notion makes it possible to achieve this goal. For example, through our design process, we have often been induced to replace a m bit serial multiplier by a n bit one (n different from m). The behavior of the multiplier being the same in both cases, we have implemented the package managing the multiplier (see figure 2) as a generic package (in the Ada sense). Whatever the number of operand bits, we have only to instantiate this generic package by the appropriate bits number. Moreover, one of the hardware design concerns is to reduce the silicon area necessary to implement a given processing. In a bit serial architecture, the variables formats have a direct influence on the silicon area. For instance, the size of the memory shown in figure 1 depends on the number of bits upon which the speech sample is coded. On the other hand, the variables formats heavily influence the sequencing. Therefore, the latter changes whenever the formats change. In order to determine the optimal bits number of a given data (e.g., the minimal number of bits giving a sufficient accuracy and thus minimizing the required silicon area) several simulations have to be made. We cannot afford to re-write (i.e., each time we change a variable format or/and the sequencing) the software representations. This is why the data upon which the sequencing depends has been implemented as generic parameters. For example, in figure 2 showing the multiplier implementation, the variable WORD_SIZE has been implemented as a generic parameter (more precisely, it depends on a generic parameter). More generally, in our experience, we have benefited from the ability of generics in Ada to have flexible programs that can accomodate a varied range of needs.

4.3. Strong typing

Earlier detection of design errors is a crucial necessity in VLSI design. Ada being a strong typed language, the compiler will act as a simulator in the sense that it will enable the detection of coherence and compatibility errors. What is more, this error detection is possible at a low cost (there is no needs to write specific software packages) and whatever the design process stage can be. For instance, in the bit serial architecture case, the communications and connections between components are done via simple wires. If therefore one defines a type -let us say- BIT, all the communications and interconnections must be defined in this term. Any violation will be automatically indicated by the compiler.

4.4. Ada and Design Methodology Interaction

When faced with a given problem, a programmer first begins to analyze the problem, then suggests one or more solutions and finally proceeds to the implementation of these solutions. Ideally, the implementation language should make it possible to faithfully reflect and to express in a direct manner the design methodology. A lot of languages do not allow the easy expression of the conventions and methodology followed by the solutions design. It is now well recognized that high level languages reduce this difficulty since they reflect more easily the definition of problems. This may explain the increasing interest for the AI languages such as Prolog and Lisp. High level general purpose programming languages are also interesting in this respect.

Our design approach can be referred to as the refinement methodology. It consists in starting from the algorithms chosen (in the first phase of our design process) and refining the descriptions progressively. The aim of this refinement is to describe more and more accurately the hardware which performs the circuit functionality specified by the algorithms selected in the first phase (software phase). In particular, some constraints associated with hardware such as the propagation delays will be included in our descriptions with more refinement. As already mentioned, the specification (and verification) of these timing constraints are made by using the CAD timing analysis tool [11]. Therefore, the design methodology we have followed is to start firstly with a top-down methodology and consists in refining the descriptions in order to gradually get closer to the real behavior of the circuit. At each stage, components models are split into more detailled lower subcomponents models. For example, a user in the speech analysis field, deals with LPC analysis shown in figure 1 frame by frame (i.e., a serie of consecutive speech samples) without making any reference to time although it is implicit in the LPC algorithm formulation. But in the real behaviour of the analysis part of the speech analysis-synthesis circuit, the signal samples arrive bit by bit at the block inputs (we have chosen a bit serial architecture). If we wish to express this fact, we must not deal any longer with the frame level but with the bit level. Therefore, we have to refine our original description (i.e., the frame level description) gradually to arrive at the bit level. To achieve this, we start by describing and simulating the LPC analysis part at the frame level without making any reference to time as usual. Then we add the timing constraints (e.g., propagation delays) by using the CAD timing analysis tool. Next, we go down to the sample level. This means that the correlator block, for instance, uses the speech samples values as they arrive to

compute the correlation coefficients. The same applies to the other blocks (e.g., lattice filter). The sampling frequency is 8 khz. We dispose therefore of 125 microseconds to do the correlation. Should this treatment need more time -considering the hardware resources available- a memorization would be needed. The size of the memory required depends on the number of bits upon which the speech sample is coded.

The next step was to describe and simulate the LPC analysis part at the bit level. We do not consider any longer that the correlation computations, for instance, are made sample by sample. Up to this stage, we have not made any reference to the hardware (target architecture) which executes the treatments. Therefore, all the steps mentioned above belong to the what we called the "software phase." When this phase is finished, we shall describe the low level hardware components (primitive elements) which perform the above treatment (i.e., LPC analysis at the bit level). We then build higher hardware components (e.g., correlator) from the low level ones (i.e., accumulator, multiplier).

As one can remark, our design work (seen as a whole) follows both a top-down approach (during the software phase) and a bottom-up approach (during the hardware phase). The latter approach is easier to implement in Ada than the former. Indeed, once the low level hardware components (i.e., primitive elements) are identified, we implement them as packages according to our implementation methodology. These packages behave as component libraries and constitute compilation units. The high level hardware components can then be built from the low level ones in a hierarchical manner as we have already explained. The "with" and "use" clauses make it possible to use the facilities provided by the low level (compilation) units. Such clauses are therefore well appropriate for a bottom-up development. On the contrary, the implementation of the top-down design methodology is more difficult since it requires that one must know beforehand the various entities he needs (e.g., subprograms and types). The first difficulty comes from the fact that often one does not know in advance in which compilation unit the entities must be defined. Therefore, the contents of the "with" and "use" clauses cannot be determined until the whole program, or at least its major parts have been designed. To conclude we can say that the "with" (and "use") clauses which determine the compilation order are helpful in a bottom-up development but are not well adapted to a top-down development. This is expressed by the fact that the information flow goes mainly from the called unit to the calling unit (the direction of this flow is determined by the order in which the various units must be compiled and therefore by the "with" clause).

The difficulties related to the implementation of the top-down approach in Ada have been outlined by Rajlich [18] who has suggested a methodology to overcome them. This methodology centers around two axes: a refinement by successive steps (a stepwise refinement) and the hiding of the useless informations at each level. According to Rajlich, this methodology makes it possible to follow more easily a top-down development. However, it suffers from the fact that the control flow (e.g., intializations and control flow direction) becomes difficult to see because of the above-mentioned hiding policy. Indeed, as long as one refines (decomposes) his design in several parts, the concern to hide useless details lead to implement the major part of subprograms and variables as local entities. Therefore, the interaction (control flow) between different design parts is difficult to see. Within the VLSI description framework, it is essential to be able to trace the flow control at each level of description in order to detect and eleminate potential timing errors.

5. TASKING ISSUES AND EXPERIENCES

Two reasons have encouraged us to develop simulation models based on the Ada tasking concept. The first one, and the most important one, is that we wanted our behavioral model to describe as faithfully as possible the real behavior of the circuit considered. Indeed, the real behavior of a hardware system consists in several entities which evolve in parallel with cooperation phases. The Ada tasking model allows one to represent both the activities performed by the hardware components and the cooperation mechanism between these activities. Each hardware component may be modeled as a task (in the Ada sense). The activity of a given hardware element is thus represented by the body of the task associated with it. As for the communications between hardware components, they can be modeled as task entries. The occurrence of a Rendezvous between two given tasks means that the communications are established. Each task is assumed to be executed on its own processor even though it is not what happens when the simulations are made on a monoprocessor machine.

The second reason which has led us to develop Ada task-based simulation models is to gain experience with the use of Ada tasks in order to see their advantages (possibilities) and potential problems associated with their use.

As already mentioned, Ada makes it possible to model the parallelism inherent in the many problems faced in practice. This fact is well known but the kind of parallelism which can be expressed by Ada tasks is not as well known. One of the possible classifications of parallelism concerns the level of granularity at which it operates:

- microscopic level: the parallelism is extracted (e.g., by the compiler) only at the level of instructions.

- macroscopic level: there is an explicit decomposition (by the programmer) of a program in "big" pieces of code which procede (often virtually) in parallel.

These definitions having been made, it is clear that the parallelism expressed by the Ada tasks belongs to the second category (i.e., macroscopic parallelism). A programmer decomposes its model into a set of processes (i.e., tasks) evolving in parallel. This kind of parallelism makes it possible to develop complex task-based models. It is usually found in Artificial Intelligence systems. On the contrary, pattern recognition machines often exhibit a microscopic parallelism.

The experience we have gained with the use of Ada tasks may be divided into two main phases. In the first phase, we have not made reference to (physical) time when using tasks (e.g., no use of the delay or select statements). The results of this (first) experience have been encouraging:

- the Ada tasks provide a good means to model components evolving in parallel (e.g., hardware components).

- Ada tasks can be used with ease.

- the difference in executing time between the programs using tasks and the others which do not use tasks is not significant.

As for the second phase of our experience, we thought that the use of statements such as "delay" should make it possible to specify the timing constraints (e.g., propagation delays) in order to analyze the timing behavior of our circuit. However, we quickly realized that such statements are not well adapted to our problem. Indeed, Ada has been designed with real-time applications in mind. These kinds of applications involve a physical time notion. The Ada mechanisms involving time (e.g., delay and selective wait statements) are therefore intented to real-time applications. On the contrary, the time concept found in the simulation models is usually a virtual time concept (e.g., advance of the simulation clock by a unit step). Furthermore, the delay statement provides a delay for at least the length of time given by an argument of type DURATION. A task is therefore reactivated (i.e., resumes its execution) at the earliest after the duration given by this expression. But, in many simulation problems, such as the simulation of a VLSI circuit, one needs to know not only what process is to be performed but also at what moment? The original mechanism of Ada Rendezvous is not directly applicable to implement simulation models such as ours which involve a virtual time concept. Moreover, in order to debug simulation models based on tasks, we need to know, at each simulation time, the processes which are active. At a given simulation step (in the virtual time scale) we may have several hardware components which are active. We need to know at which moment in the virtual scale time the processes which are resumed and the current running process. But in Ada, the task execution model is a pseudo-parallel one, i.e., the processor is automatically attributed to various processes without any interference of the programmer, unlike the case of Modula-2 [22] in which we have a quasi-parallel execution model (there is no procesor sharing mechanism). This is expressed by the fact that in Modula-2, the programmer must specify the process (referred to as co-routine) which is to be resumed: Transfer(p_1, p_2), in which p_1 designates the process to be suspended and p_2 is the process to be resumed.

For problems like ours (i.e., hardware components descriptions), we thus need a low level tasking scheduling mechanism which looks like the one in Modula-2. On the one hand, one must implement the virtual time concept and on the other hand, one must know the order in which the tasks modeling the hardware components are managed (scheduled). To achieve this goal, we have used the discrete event simulation technique known in the literature as the process interaction approach [8]. This approach relies on a set of processes which performs the calculations (operative part) and on a scheduler (control part) which will schedule tasks according to their activation times (i.e., awakening instants). The implementation of this approach in Ada has consisted in identifying both the scheduler and software representations of the hardware components as Ada tasks.

In order to schedule a given task in the appropriate order, we let it wait for a Rendezvous. At the appropriate moment (in the virtual time scale), tasks can be unblocked because their entries are ready for a Rendezvous with other tasks (i.e., their awakening instants become equal to the current value of the simulation clock). The scheduling mechanism we have developed therefore uses the original Ada mechanism. But unlike the Ada original mechanism, the order of tasks scheduling is known by the user in our scheduling mechanism. The scheduling mechanism is "no longer transparent to the programmer."

The simulations which have been made with these Ada task-based models have raised the following problems:

- it is difficult to determine the exact causes of the deadlocks which have been observed since our simulation models involve a great number of tasks, mainly at the bit level description. Consequently, the debugging of programs is a time-consuming task.

- we have observed an important overhead at the bit level description compared to the approach in which the scheduler and the software representations have been identified as Ada subprograms. This overhead is mainly due to the communication times and to the sequential task access to hardware resources (we have used a monprocessor machine). The switching time depends on the underlying (i.e., host) machine and seems to be poor on a VAX 11/750 under VMS version 4.6 and with a DEC compiler version 1.4 (arround hundred of microseconds) while it seems to be about 60 nanoseconds on current machines. In the version 1.4 of the Ada DEC compiler, some components (e.g., debug and trace) have been included with the VMS operating system and changes in the return mechanism along with an input caching mechanism have been introduced to enhance performances. VMS version 4.4 has included run-time tasking performance improvements; in particular Rendezvous time has been sped up. However, the problems mentioned above (and in particular, that of poor switching time context) still remains.

In the particular case of a target architecture description which involves a slow traitement like the bit serial architecture we have chosen, one can conclude that the overhead relevant to the use of tasks is important. Indeed, in this case, a tasking-based simulation model would imply a large number of Rendezvous and thus a big number of commutations. The approach which uses Ada task-based models have been therefore given up. We then adopted the other main discrete simulation technique, i.e., the event scheduling approach [11]. The Ada implementation of this technique rests on Ada subprograms-based models. To be more precise, the scheduler has been implemented as a procedure (in the Ada sense). As for the software representations of hardware components, they have been identified as subprograms, as shown in figure 2 relevant to the bit serial multiplier. In other words, the scheduler and the operative part (i.e, software representations) have been implemented as Ada subprograms, unlike the previous approach in which they have been identified as Ada tasks. In the former approach, the procedures behave as passive tasks and they can carried out simultaneously [11]. The choice of discrete event simulations techniques in our experience has several advantages, as argued in [11].

Distributed scheduling is more appropriate for the tasking model. Indeed, simulation models can be executed faster on a multiprocessor architecture. But in fact, many problems arise. These problems are linked partly with parallelism semantics and management. We hasten to add that most of these problems are not directly related to Ada but mainly with parallelism. For instance, it is difficult to define and to

implement a consistent notion of time across a distributed system. Generally, in such an environment, the notion of a global clock does not exist. With Ada, it is difficult to define the delay statement semantically. Performance is another problem. Indeed, the cost associated with parallelism management can be significant. Most of present runtime environments associated with compilation systems do not support distributed configurations efficiently. This could explain why some users try to change the runtime libraries in order to obtain the functionality and the performances desired. Nevertheless, such work is complicated and long-lasting. Even more serious is the fact that the obtained compilation system can become invalid. On the other hand, different kinds of problems can appear, like the ones raised by test and checking (e.g., determination of the task state and of the state of calculations). Furthermore, the problem of distributed termination arises (the termination information is distributed among processors). So, each processor must be able to know that the program is over.

The problem of tasks management with timing and resources scheduling constraints is another kind of problem. A lot of algorithms suggested are NP-Complete algorithms. We also lack quantitative and qualitative criteria for the evaluation of parallel algorithms. Finally, with this kind of problems associated with the use of tasks on a distributed systems, we can mention the problem which arises when two tasks are already engaged in a Rendezvous if processor failures have to be tolerated. Indeed, the select and exception clauses do not allow one to solve this problem [13].

As for the user, the inefficiency of support tools may cause debugging and performance problems. Likewise, the tracing of control flow may be difficult. Finally, we could mention that Ada does not have syntaxic or semantic structures available to enable it to assign a physical processor to the data structures which represent a process (task) ready for execution. Consequently, one can expect that Ada tasks management in a distributed environment can be transparent to the programmer or will be achieved with synchronization primitives outside the language. The drawbacks of the second alternative are obvious. As for the first one, we have seen that when task management is transparent in a centralized environment (i.e., in principle the programmer is unaware of its existence), some problems arose at which a user needs to know the order in which tasks are scheduled and the time when these tasks are scheduled. One might expect that we could face the same problem in a distributed environment.

6. RELATED WORK IN HARDWARE DESCRIPTION

Since the advent of Ada, several studies have been made regarding its use in hardware description. These can be classified in two main categories. The first one consists in translating hardware descriptions written in Ada to silicon (e.g., Girczyc [10], Organick [17]). The second one only uses Ada as a hardware description language (i.e., Barbacci [2], Ghosh [9]). The following comparisons will be based on the second type of approach.

Ghosh's approach is based on the Ada tasking model. Each type of component is modeled as an Ada task type. Task types constitute generic models (not in the Ada sense). The component type instances correspond to the task instances. The Ada model for a signal path consists in a a (record) message between instances. The timing characteristics of hardware elements such as propagation delays are modeled as a field of the record message. A large degree of scheduling is distributed among the model instances. A task schedules itself for execution when all necessary inputs are asserted at its input ports. The underlying scheduling of the tasks is a part of the Ada runtime support. On completing execution, the output generated are communicated directly to all other tasks that are connected to its output ports. Connectivity is modeled as an interconnection data base. Nevertheless, this kind of approach based on Ada tasks has drawbacks as we have explained (e.g., simulation models involving a large number of tasks are difficult to develop and to debug and may result in a execution overhead). Moreover, unlike our approach, there is no genuine timing analysis tool. This last drawback also applies to the Barbacci's approach since time is modeled the same way as any other procedure parameter and there is no explicit synchronization core (no scheduling mechanism). We can therefore expect that a designer in the VLSI design framework would find it difficult to detect timing errors since no timing analysis tool is provided to follow the time flow throught his design.

7. SIMULATION EXPERIMENT

Our modeling approach (i.e., each package manages a particular kind of hardware component) and our timing analysis tool [11] based on the discrete event simulaton technique known in the literature as the event scheduling technique [8] have been successfully used to describe and to simulate the analysis part of the speech analysis-synthesis circuit. Moreover, they have been used from the top level description (i.e., description of the LPC analysis at the frame level) down to bit level. The simulations have been made on a Vax 11/750 under VMS version 4.6 machine using a DEC compiler version 1.4.

Because of the limited space in this paper, we gives only two CPU times (see figure 3). The first one is related to the LPC analysis at the frame level. This example belongs therefore to what we referred to as the software phase (simulations models are executed on a predefined hardware without specifying the target architecture). The second CPU time is related to the example of a hardware component (i.e., the correlator). The software representation of the correlator specifies an execution process different from the predefined one, i.e., based on the choices we have made (e.g, a bit serial multiplier).

Straight away we can notice that the hardware model consumes more time than the software model. Indeed, the software model includes the correlator block as well as two other blocks (i.e., the lattice filter and the PARCOR extractor blocks). This difference in CPU times is not surprising since the hardware model describes a target architecture and it is constrained both by the algorithms and the hardware while the software model is executed on a predefined hardware and is contrained only by the algorithms.

8. CONCLUSIONS AND RECOMMENDATIONS

An important contribution of our study was to provide an evaluation of the use of Ada in the simulation field. Our experience has demonstrated the interest of using Ada in a complex development project and in the hardware simulation domain in particular. Unlike other papers which deal with this topic on the basis of Ada's potential advantages and drawbacks, the ideas and results presented in this paper are based on the lessons learned from a real development effort.

As mentioned earlier, there is no use of a specialized simulation language. It is Ada that is used instead. There are many advantages associated with this choice:

1- advantages resulting from the choice of a high level language.

2- advantages due to the choice of Ada.

3- advantages due to the choice of one single language during the whole design process.

Regarding the first point, the need for a high level language can be justified by the fact that users in the signal processing field usually use high order languages like Fortran to implement their algorithms which are quite simple (set of multiplications, additions and divisions). Two main parts can be seen in these algorithms: repetitive data calculations and control. The choice of a low level language involves not only some difficulties when learning it, but also would make it unavoidable for these users to handle low level data they are not used to. More generally, at the first stages of design, the circuit designer must concentrate on a high level of abstraction for the description and simulation (e.g, behavioral level). The data and control handled at this level are abstract objects. So the description and simulation language must be able to implement easily such objects. A low level language does not have either such high data structures or such sophisticated control structures to do it. Moreover, only high level languages provide structures powerful enough to tackle complex problems. As far as we are concerned, we have adopted Ada as a high level language for the development of our CAD tool as well as for the software and hardware description and simulation. This choice is vindicated by the advantages of Ada over other high level languages.

The choice of Ada instead of other high order programming languages as our high level language has proved to be a good choice. Indeed, Ada makes the modeling and simulation particularly easy: powerful data typing, generics, sophisticated mechanisms for data abstraction and control abstraction, modularity, improved portability, availability of a support environment, etc.

On the other hand, the use of the same language during the whole design process makes the testing of programs and the detecting of potential errors in the descriptions particularly easy since the same language is used both for hardware and software simulations. We can then benefit from existing development and debugging tools (there is no need to develop another compiler or another debugger). In the case of Ada, the code is reliable and the testing is easy since the debugging environment is powerful. There is also no need for the designer to learn another language. Moreover, the decision to implement the components in software or in hardware can be postponed to the very last moment. However, the use of a general purpose programming language like Ada for hardware description may have some drawbacks. Indeed, this type of languages lacks appropriate timing primitives (a virtual time concept) and predefined structures (e.g., register, bus). Our approach for the hardware description and simulation presents several advantages over the existing ones, especially when it comes to the time modeling and specification. Indeed, the approaches using Ada as a language for hardware description and simulation found in the literature (e.g., Barbacci [2]) do not present any methodology for the specification of temporal constraints for hardware components. For instance, the approach adopted by Barbacci [2] at Carnegie-Mellon University specifies temporal constraints (e.g, propagation delays) as parameters to the procedures implementing the hardware components. On the contrary, we have shown that the specification of these temporal constraints has been achieved by means of a well defined and elegant technique (i.e., event scheduling technique). On the other hand, our approach for hardware description is also different from the one taken by Shahdad. Indeed, we have stuck to the Ada language whereas Shahdad [20] has developed a new language based on Ada called VHDL (a part of the VHSIC program) dedicated to hardware description and simulation. Moreover, we have used the same language (i.e., Ada) for hardware and software simulations whereas the VHDL language is used for hardware descriptions only. This has several drawbacks since it sets up a border between a software designer and a circuit designer. It cannot benefit from the advantages connected with the use of a single language during the whole design process (e.g., use of the existing development and debugging support tools).

On the other hand, our experience has proved it necessary to use high level descriptions (i.e., functional descriptions) along with a high level language in order to design and simulate an architecture quickly and reliably. The choice of Ada and of a functional level of description have proved to be good. As for this later point, the choice of a functional or behavioral level in opposition to the structural level (logic, gate, circuit and switch levels) can be justified by the fact that the model size of the circuit increases as the design progresses (i.e., when we the descriptions are refined). The complexity of the evaluations increases therefore in an unlinear way. To overcome this difficulty, one solution is to use functional descriptions implemented in an appropriate language. A high order language is then desired. A functional description does not consider the underlying logic. It only takes into account the input-output dependence relationships and the propagation delays implied in this dependence. Moreover, the verification is faster. This explains why our CAD tool enables one to specify in particular the propagation delays which are implemented as fields of the records associated with events.

The question is whether the use of Ada for hardware (and discrete event) simulations results in an overhead with respect to the specialized languages. In particular, compile-time and runtime checking may lengthen the execution times. It seems that no exhaustive study have been made on this topic. This could be explained by the fact that the potential overhead of Ada compared to other languages depends on the compilers efficiency, on the application case and on the host machine. It is not sure that the use of some pragmas such as inline or optimize enable to reduce the execution times. On the contrary, one might fear that the use of

such pragmas may cause CPU times to increase. Our experience corroborates this fear. Indeed, we have observed on the lattice filter example that the CPU time increases when we use the pragma optimize: 5 seconds instead of 4 seconds (no use of pragma). This could be explained by the fact that when this pragma is accepted by the compiler, the compiler has to adopt choices which are not "optimal" compared to the choices it would have made (choices which take partly into account the host machine). The use of the suppress_all pragma has not decreased significantly the CPU times observed.

One of the main points discussed in this papier concerns our experience in using Ada tasks. In particular, the use of Ada tasks to model complex systems such as an integrated circuit has arised the need for efficient compilers and support environments. To achieve such efficiency, one can think of designing machines based on an architecture, part of which would take into account a few Ada semantics and especially the ones relevant to tasks. Indeed, Ada task-based simulation models can be potentially executed faster on a distributed system. Nevertheless, we have stressed the problems raised by the verification, testing and debugging of Ada tasking programs in such an environment. Advanced tools for the testing and debugging of concurrent Ada programs are therefore needed (both in a centralized and distributed environments). However we do not claim that the use of Ada tasks on a monoprocessor machine should be avoided in the case where memory size and execution times are significant factors, provided that powerful support tools are available. Many papers have been published which deal with the implementation issues and with the use of Ada on a distributed system [13], [23].

Another main characteristic of Ada we have mentioned concerns generics. In this respect, we have shown how the use of this facility makes it possible to have flexible programs and to reduce their sensitivity to changes. This flexibility is all the more interesting as the design and simulation are repetitive processes. Furthermore, the description of a complex system such as an integrated circuit involves several thousands lines of code. However, it is clear that in the case where efficiency have to be achieved, simulation models must take into account the hardware types supported by the host machine. The code is consequently non generic ("dedicated" to the host machine).

Besides, our design methodology has several advantages. It looks like the structured design methodology suggested by Mead and Conway [15] at Caltech University. This methodology is consistent with Ada philosophy since it is based on two main characteristics: hierarchy and regularity. The hierarchy can be achieved through nested Ada packages which represent the split up of the design into more refined components. In our application, it consists in the gradual refining of the description. This structured design methodology is similar in concept to structured programming [10]. It enables one to implement design more quickly and more reliably and to prove the correctness easily. However, we have seen that the implementation in Ada of our design methodology corresponds to a top-down development and brings about some problems, since the information flow goes mainly from the called unit to the calling unit.

To summarize, from our exprience, we feel that Ada provides valuable constructs and mechanisms for modular design of complex software and hardware components. Ada presents several advantages over both specific simulation languages and general purpose programming languages even though its use for some simulations problems may have some drawbacks (e.g., lack of a virtual time concept and of a few structures appropriate to hardware design). However, these deficiencies seem to be minor compared to the gains related to the use of Ada. Indeed, our experience has thrown into relief several advantages resulting from the use of Ada. After an investment in the phase of learning Ada (3 months), we observed that our productivity increased day after day.

We cannot conclude this paper without mentioning an interesting contribution to the study of the Ada impact on software cost, quality and productivity made by Reifer [19]. His study concerns 41 projects which delivered over 15 million lines of Ada code for a variety of applications, mostly real time. The results of his data analysis show the following:

- effort distribution was different.
- productivity was better.
- error rates were lower.
- required development resources were less.

Regarding the first point, Reifer'study has demonstrated that Ada developments tend to observe a 50:15:35 distribution (i.e., 50% of the software effort is allocated to requirements and design, 15% to development and 35% to testing). This is to be compared with the distribution of traditional projects (i.e., 40:20:20). Another main fact is that 30 among the 41 projects surveyed use object-oriented design as their detailed design methodology. This is close to the methodology followed for the Colombus project. Moreover, 36 among the 41 projects also used Ada as a design language. This emphazises the idea that high order languages in general and Ada in particular narrow the gap between conception and implementation methodologies.

As to the productivity merely defined as being the number of Ada source lines of code (ASLOC's) per person-month of effort made, Reifer's study has shown that average productivity during software development was 310 lines Ada source lines of code per person-month (i.e., 310 ASLOC's/person-month). This compares nicely with an industry average of about 200 SLOC's/person-month of effort. However, on the average, productivity improvements were not achieved until the third project was completed by the project team. As to the third point (i.e., error rates), the average error densities observed was 3 to 5 errors per KASLOC. This compares quite favorably to the Air Force's experience of 4 to 13 errors per thousand lines of code.

Lastly, for the fourth point, Reifer'study can demonstrate that as Ada projects get bigger, they get cheaper. This phenomenon challenged the power laws that most of the popular software cost models were based upon which say that as a size gets bigger, time and effort increase according to a log-log relationship because of the management burden associated with inter-group communications. More precisely,

the basic equation used by most code models [1] for estimation effort is the following:

$$effort = A(size)^p \quad (4)$$

where A is a constant technology and p the power law (describes the relationship that exists between size and effort).

While most cost models assume that this power is greater than one, the Reifer's study indicated that for Ada projects, this power law stabilizes at 0.95. This can be interpreted to mean that as projects get bigger, the opportunity for reuse is so great that it damps out the management burden to make larger projects cheaper. Reifer's study also proved that current compilers really cannot adequately handle real-time tasking and Rendezvous. As for the question of Ada cost model development, Reifer's paper shows that the use of object-oriented methodologies and the performance of the Ada compilation system have much greater effects on productivity than in classical systems. Equally, the degree of real-time and system architecture tend to be more important factors driving Ada costs because of the difficulties in designing and mechanizing Ada tasking.

Despite the fact that the model validation used by Reifer to issue his statistics is not a mathematical validation (the approach taken by Reifer to validate the accuracy of the prediction of Ada software cost consisted in comparing the model's estimates against actuals taken from completed projects given to him by customers), his study seems to be the most exhaustive one on the Ada impact on software cost, productivity and quality. Reifer gives several recommendations for development expriences. In particular, it seems that productivity can be improved by at least 25% .

References

1. Bailey, E. K., Frazier, P. and Bailey, J. W. A Descriptive Evaluation of Software Cost-Estimation Models. Institute for Defense Analyses, Paper No. P-1979, October 1986.
2. Barbacci, M. et al. Ada as a Hardware Description Language : An Initial Report. Proceedings of the IFIP 7th International Symposium on Hardware Description Languages and their applications, Tokyo, August 1985. North-Holland Publications.
3. Batali, J. The DPL/Daelus Design Environment. Proceedings of the VLSI 81 Conference, Academic Press, 1981.
4. Berry G., Moisan, S. and Rigault, J. P. ESTEREL: Towards a Synchronous and Semantically Sound High Level Language for Real-Time Applications. Proc. IEEE Real-Time Symposium, 1983.
5. Caspi, P., Halbwachs, N., Pilaud D. and Plaice, J. A. LUSTRE: A Declarative Language for Programming Synchronous Systems. 14th ACM Symposium on Principles of Programming Languages, Munich, Janvier 1987.
6. Chappell, S. G., Menon, P. R., Pellegrin, J. F. and Schowe, A. M. Functional Simulation in the LAMP System. Journal of the Design Automation and Fault Tolerant Computing, Vol 1, No 3, MAY 1979.
7. Dahl, O., Myhrhaug and Nygaard, K. Simula 67 Common Base Language. Publ. N0. 5-22, Norwegian Computing Center, 1970.
8. Fishman, G. S. Concepts and Methods in Discrete Event Digital Simulation. A Wiley-Interscience Publication, John Wiley and Sons, 1973.
9. Ghosh, S. RDV : A Rule-Based Generalized Design Verifier. Ph.D Thesis, Department of Electrical Engineering, Stanford University, Stanford, CA, 1984.
10. Girczyc, E. M. Automatic Generation of Microsequenced Data Paths to Realize Ada Circuit Descriptions. Ph.D Thesis, Department of Engineering, Carleton University, July 1984.
11. Guennouni, J. Simulation and Temporal Verification of a VLSI Circuit Using Ada: A Case Study. To appear in the Summer Computer Simulation Conference proceedings, 25-28 July, Seattle 1988.
12. Hill, D. Multi Level Simulator for Computer Aided Design. Ph.D. Thesis, Center for Integrated Systems, Computer Systems Laboratory, Stanford University, CA, 1980
13. Knight, J. C. and Urquhart, J. I. A On the Implementation and Use of Ada Fault-Tolerant Distributed Systems. IEEE Transactions on Software Engineering, Vol. SE-13, No. 5, May 1987.
14. Markel, J.D. and Gray, AH. Linear Prediction of Speech. Communications Cybernetics 12, Springer-Verlag, 1976.
15. Mead, C. A. and Conway, L. A. Introduction to VLSI Systems. Adison Wesley 1980.
16. Mouly, J. C, Neirynck, J. and Tarpin, F. Prolog Based CAD tools for VLSI. Proceedings of the 1987 European Conference on Circuit Theory and Design, Ecole Nationale Supérieure des Télécommunications, Paris, France, Sept. 1-4, 1987, 639-644.
17. Organick, E., Ogilvie, J. W. L. and Henderson, T. C. Signal Processing-to-Silicon using ADA as a Hardware Specification Language: An initial Investigation. Report of the Department of Computer Science, University of Utah, SALT LAKE CITY.
18. Rajlich, V. Refinement Methodology for ADA. IEEE Trans. on Software Engineering, Vol. SE-13, No-4, April 1987.
19. Reifer, D. J. Ada's Impact: A Quantitative Assessment. Proceedings of the 1987 ACM SIGAda International Conference on the Ada programming language, Boston, December 9-11, 1987, 1-13, ACM SIGAda publications.
20. Shahdad, M., Lipsett, R., Marschner, E., Sheehan, K. and Cohen, H. VHSIC Hardware Description Language. Special issue on Computer Hardware Languages, IEEE-CS Computer, Volume 18, No. 2, February 1985, 94-102.
21. Schriber, T.J. Simulation using GPSS, Wiley, New York, 1974.
22. Thalmann, D. Modula-2: An Introduction. Springer Verlag, 1985.
23. Volz, R. A. and Mudge, T. N. Timing Issues in the Distributed Execution of Ada Programs. IEEE Transactions

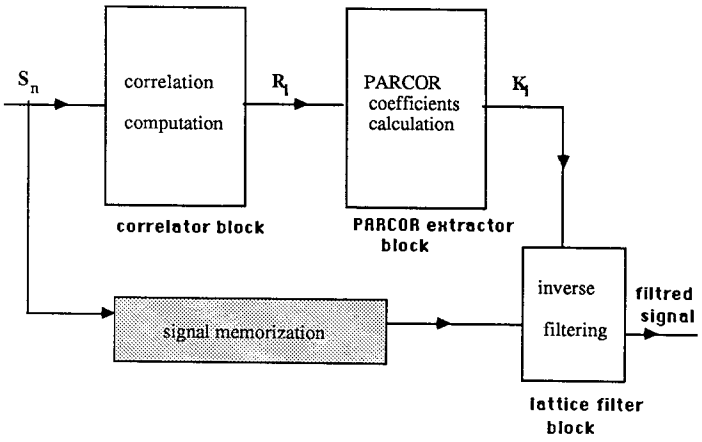

figure 1: LPC analysis

```
with BITS_HANDLER      ;    use BITS_HANDLER ;
with  BIT_SERIAL_ADDER ;  use BIT_SERIAL_ADDER ;

generic
       NUMBER_BITS_OF_THE_FIRST_OPERAND      :  natural ;
       NUMBER_BITS_OF_THE_SECOND_OPERAND     :  natural ;



package BIT_SERIAL_MULTIPLIER   is

     subtype PRODUCT_BIT is natural range 0 .. 1 ;
        ...
        ...


     procedure MULTIPLY_BIT_SERIAL(.......) ;

 end BIT_SERIAL_MULTIPLIER ;



package body BIT_SERIAL_MULTIPLIER is
     ...
     procedure MULTIPLY_BIT_SERIAL(.......) is

          ...
          begin
             LOAD_FIRST_OPERAND ;
             EXTEND_MULTPLICAND_BY_ONE_BIT ;
             MULTIPLY_BIT_BY_FIRST_OPERAND ;
             SHIFT_PRODUCT;
             ACCUMULATE_SHIFTED_PRODUCT_AND_PARTIAL_SUMS;
             EXTEND_PARTIAL_SUMS_BY_ONE_BIT;
             ...



     end MULTIPLY_BIT_SERIAL ;

 end BIT_SERIAL_MULTIPLIER ;
```

model	CPU times
software model of the LPC analysis	9 seconds
hardware model of the correlator block	11 seconds

figure 3. CPU times for the LPC analysis and the correlator block

these measures are made on a VAX/VMS version 4.6 with a DEC compiler version 1.4 and concern 800 speech samples

formats

- speech sample represented by a word of length 12 bits
- correlations coeficients represented by word of length 25 bits