On the use of VHDL–based behavioral synthesis for telecom ASIC design.

Mark Genoe **Paul Vanoostende** **Geert Van Wauwe**

Alcatel–Bell, Advanced CAD for VLSI
F. Wellesplein 1, B–2018 Antwerpen, Belgium

VHDL–based behavioral synthesis is appearing on the market but it still has to prove that it can have a significant impact. In the past, most applications for behavioral synthesis came from the DSP area and from the academic world. In contrast, this paper describes the results of an investigation and evaluation of several behavioral synthesis tools, carried out on recent designs of Alcatel–Bell, leading to a more detailed study of relevant industrial telecom non–DSP circuits, that were suitable for behavioral synthesis.

From our expertise in telecom system hardware design, we can conclude that, taking into account that today world–wide about 6,000 licenses for logic synthesis are in use, there is distinctly a market potential for design–entries at higher levels of abstraction, due to the still increasing design complexities that can be expected in the near future. Behavioral synthesis can play a key role in this prospect, as stand–alone hardware CAD tool, or integrated in a global system design flow strategy for HW/SW–codesign. However, we experienced that efficient use of behavioral synthesis tools for telecom non–DSP circuits requires functionality that goes beyond simply generating an RTL–synthesizable description. This functionality is discussed, together with a system level design methodology for efficient use of behavioral synthesis tools.

1. Introduction.

Electronic design has moved since long from transistor–level to gate–level design, followed the last few years by a second transition from schematics to register–transfer (RT) level design entries. Examples of common–use commercial products in this context are the Synopsys Design Compiler, Mentor's AutoLogic, ViewLogic's ViewSynthesis, Cadence's Synergy, and Exemplar Logic's Core Synthesis System. Behavioral synthesis has been an active field of research for about one decade [1][8], leading in particular to a number of academic demonstrators (see e.g. [2], [5]). The main trend towards higher and higher levels of abstraction has one and the same origin, and is heavily influenced each time by smaller technologies and growing complexities of IC's

Today, designers are confronted with very time–consuming description efforts at the RT–level, including debugging and design–iteration activities. Due to the fact that time–to–market is a key issue, many designers are looking for new software tools to bridge the gap from a register transfer to a behavioral entry level. Currently, based on earlier expertise from mostly academic research, a number of commercial behavioral synthesis tools are appearing [3]. Examples of this first generation of commercial behavioral synthesis tools, are e.g. Behavioral Compiler from Synopsys Inc., the Mistral–2 DSP Compiler of Mentor Graphics Corp., the SYNT compiler of Synthesia AB of Sweden, and the LAMBDA system design tool of Abstract Hardware Limited. However, they still have to prove that they can have a significant impact, i.e. that they reduce the design time for a significant portion of an average design.

In the past, confrontation with industrial practice has often led to unexpected applications for CAD technologies. We believe the same holds for behavioral synthesis. The purpose of this paper is therefore to describe the lessons we learned during initial experiments with behavioral synthesis for our telecom applications. These applications are non–DSP, which means that they don't process signals, but consist of actions like bit manipulations, counting, detecting specific word–values, storing/retrieving datawords etc.

More specifically we will discuss the following questions:

- For which telecom applications can commercial behavioral compilation tools be used, with the current state–of–the–art? What additional functionality is needed to handle ”real–life” ?
- What functionality does the user need to control and understand the results of the synthesis ? Which trade–off can be made between automation and necessary interaction ?
- Where and what do we gain by applying behavioral synthesis in the design flow strategy, compared with figures of a more traditional register transfer approach ?
- What is the next step after behavioral synthesis, and how does this fit into a global strategy for future hardware–software heterogeneous systems ?

Our conclusions follow from an investigation of several designs that have been implemented at the RTL–level and that are possible candidates for implementation with behavioral compilation. Typical complexities considered are about 10 to 30 K gates of logic (excluding RAM).

Our initial impressions on VHDL–based behavioral synthesis for telecom ASIC design have been reported in [9]. These impressions have been consolidated over the past months, and are described in this paper. An important objective of this presentation is to spawn a lively interaction between academic researchers on advanced synthesis, commercial CAD companies providing this functionality, and potential users from system houses.

The paper itself is organized as follows: section 2 describes the traditional design flow, and indicates the disadvantages of this approach for current design complexities. In section 3 we present a typical telecom application, and illustrate the impact of behavioral synthesis for such designs. In section 4, we describe a number of premises, additional requirements and improvements for behavioral synthesis. In section 5, the most important features that can be expected from next generation system synthesis tools – beyond behavioural synthesis – are summarized, which will extend largely the productivity gain.

2. Why there is a definite need for behavioral synthesis.

Traditional design entries (at RT–level) cannot deal any more with complex systems, due to the extremely huge effort to describe the desired functionality at the entry–level.

2.1. Traditional design entry

Traditional design entries become more and more restricted, but can still be used. The RT VHDL design–entry and the graphical state–diagram entry are two of them:

- RT VHDL design–entry : for a block whose behavior is roughly the same for each clock cycle, traditional RT–based logic synthesis remains the optimal design–entry paradigm. This behavior can be influenced by a state, but the main part of the blocks consists of a number of pieces of combinational logic that are separated by flip–flops. The complexity of the block is dominated by this ”*data–pipeline*”.
- Graphical state–diagram entry (e.g. [6]) : for a block whose behavior depends on its state, and whose state–transition diagram does not contain a dominant linear flow. Although it is not easy for a synthesis tool to subdivide the flip–flops of a design into flip–flops that store state–information and flip–flops that store data values, the designer usually makes a conceptual distinction between them, when he is designing a *FSM*. Only a few flip–flops store state–information : the number of states is typically less than 50, and the number of data flip–flops is hence much larger than the number of state flip–flops.

There is no doubt that state–diagram synthesis and even more RTL synthesis have provided already an enormous productivity gain in the last few years with respect to earlier schematic–entry approaches. However, with the still increasing silicon technologies densities (0.35 micron ...), resulting in a continuous multiplication factor of about 1.2 each year for the total amount of transistors and functionality on silicon, combined with the still growing influence of the time–to–market aspect for new IC’s in telecom on the sales profit, abstraction to higher levels is mandatory in the near future. This will result in ”easier” functional design descriptions, without detailed control aspects, and tools which can perform an exploration in the design space, with essential tasks such as scheduling, allocation and resource sharing across multiple clock cycles.

2.2. Main problems with the current approach

To implement a design that implements a multi–cycle algorithm, the designer today typically writes two kinds of VHDL, one for the data computation part and one for the sequencing part :

For the computation unit, the VHDL code typically describes the structure, consisting of datapaths, registers, busses or multiplexers, augmented with a behavioral description per datapath (e.g. an adder). To develop such code, an analysis is needed of the operations that must be performed (e.g. finding the first zero in a bitvector). The resulting code contains a limited number of VHDL–lines, is structured, and easy to become re–acquainted with during redesign.

For the sequencer, the VHDL code typically has the following structure :

```
case Counter is
    when 10 =>
        if mode = do_command1 then
            connect_regA_ramAddressPin <= ’1’
        ...
    when 11
    ..
```

To develop such code, the elementary actions (e.g. write–to–memory) must be scheduled, and, for each control–step, the micro–actions (e.g. set a register load signal to true) must be written explicitly. Writing such code is error-prone. The resulting code contains a lot of VHDL–lines that describe the design with high detail. They are hard to debug and error–prone, and don’t allow reusability. Redesign of such code is particularly difficult : the code is hard to understand, and rework.

As a consequence, we believe that *the main potential for behavioral synthesis lies in avoiding straight implementation of the controller.* In addition, synthesis can help during the design of the processor, by requiring from the user only the definition of the key features of the processor architecture (e.g. the application–specific datapaths) and by optimizing the resource usage.

The most important synthesis steps that can be found in commercial behavioral tools to bridge the gap between the register transfer description and a behavioral description, are the following (see also [8]): *dependency graph construction, resource allocation, scheduling, register assignment,* and *datapath interconnection.* These tasks allow fast exploration of the global design space in contrast with the traditional approaches, especially for re–design purposes. However, with current state–of–the–art commercial tools, restrictions exist still in e.g. the computational complexity size of the algorithms to implement, in describing and implementing multiple entities and processes, and in defining complex pipelined designs.

3. Behavioral synthesis applied on industrial telecom designs.

In telecom, the digital circuitry can be roughly subdivided into 3 classes :

- logic that implements the interface with the on–board controller or that stores the state of the ASIC (e.g. the number of errors of a certain type).
- logic that is basically doing the same job each cycle, e.g. a scrambler/descrambler.
- logic that starts processing once every X clock cycles. This processing takes a certain number of clock cycles. Note that telecom data is typically structured into cells or frames, e.g. the ATM cell of 53 bytes, and the SDH STM1–frame of 2430 bytes. Per cell or frame, a number of tasks must typically be performed.

We believe that it is this last class that is suited for implementation with behavioral synthesis. It is considered in the sequel. Inside this class, datapath–dominated DSP–systems have been the main focus of research (see e.g. [5]). However, as mentioned above, our applications are mostly non–DSP and we investigated the applicability for behavioral synthesis outside the DSP area.

3.1. A typical telecom design example

The Alcatel Connectionless Transport Server (ACTS) architecture [4] provides the necessary connectionless service functions for the direct provision and support of e.g. switched multi–megabit data services (SMDS) in a B–ISDN environment through a high–speed ATM–cell based transport network. In order to meet the requirements, all packets (ATM–cells) have to be switched on segment/ cell basis, and their appropriate functions have all to be executed in a cell–time unit. In its current implementation, the ACTS consists basically of three ASICs. The Preventive Congestion Controller (PCC) is one of them. It is used to control the resource allocation for out–going link connections, such as MID and bandwidth allocation.

The MID–manager block keeps track of the message identifiers (MIDs) that are allocated by a user of the considered telecom system. Each user can allocate up to 1024 MIDs. The status of each MID is represented by one bit in a bit–map arranged as a matrix of 32 rows by 32 columns. Each row is stored in one word of an external memory array. When a MID is requested, one must scan the bit–map to find a free MID. To reduce the number of memory accesses, the bit–map is augmented by a summary word (see figure 1). Each bit in this word gives the status of the corresponding row, i.e. if all MIDs in this row are allocated or if at least one MID is still free. When a MID is requested, the summary word is read and one searches for a bit equal to zero. The column number of this bit gives the row number of a row containing a free MID. This row is read and one searches a bit equal to zero. This algorithm can find a free MID using two reads and two searches. After the allocation, the row is updated, and, if needed, the summary is updated.

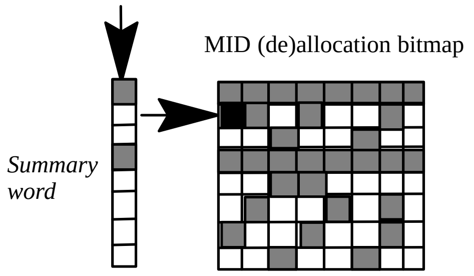

Figure 1 MID allocation principle

Each user has two bit–maps. A bitmap can be active or inactive. When a bit–map is active, it is used to allocate and deallocate the MIDs. The two bit–maps can be both active. In this case, a MID can be allocated if and only if it is marked as being free in the two bit–maps. This can be implemented easily by taking the bit–wise OR of the two summary words and of the two selected bit–map rows.

The reason for two different bit–maps for each user is garbage collection. Normally, the MID's are always deallocated by a MID deallocation request. However, such a request can be lost in the switch fabric or under abnormal conditions it is not generated at all. So, some MID's can be lost, i.e. never deallocated. The goal of the garbage collector is to bring them back into the free pool of MID's. The proposed method takes into account that the packet life–time is bounded. By continuously switching the bitmaps from active to inactive – and vice–versa – , and by re–initializing each inactive bitmap after having reached the maximum life–time, the negative effect of losing MID deallocation request can be solved.

This example highlights what we believe to be typical characteristics of non–DSP telecom applications for behavioral synthesis :

- the behaviour contains usually already a number of ”high–level” state machine descriptions, such as e.g. the state of the bitmaps (active, reset, idle, ..) and the resulting actions in the PCC–example.
- the data throughput (cells, timeslots, frames, ..) is in most of the applications rather high, which results in a rather small amount of available clock cycles (typical 10 – 100).
- at the other hand, the input–output delay of the data is not that crucial. This allows synthesis of different pipelined sections.
- the most critical resource is often the RAM. The RAM–accesses are typically the limiting factor when sched-

uling an individual block. Often several blocks share the same data, which complicates the synthesis task.

3.2. Behavioral synthesis results

The algorithmic description for the PCC–chip can be summarized as follows: within each frame of 53 cycles at 20 Mhz, two allocation/deallocation commands have to be processed. Normally two bitmaps have to be updated for each command, except when one of them is in reset–mode (garbage collection). The most frequent operations of the algorithm are specific memory address computations and accesses, and a number of bit–manipulation functions (such as finding a first zero bit in a 32–bit word; changing the value of a single bit of a 32–bit word; concatenation of several fields in order to obtain correct RAM addresses, etc.)

3.2.1. Behavioral VHDL versus RT–VHDL

VHDL is a strongly typed general language (meaning that every object must be declared before use) intented mainly for simulation and documentation purposes, in order to improve the specification and validation process of complex integrated circuits. Predefined types and conversion rules are declared in Standard packages, but new types can be defined based on them. VHDL is being used to support a number of automated CAD synthesis tools.

To provide a basis for the discussion, the basic functionality of VHDL–based behavioral synthesis is shown in Figure 2. We like *the use of VHDL as design–entry language*, because VHDL has most constructs required for a behavioral description and the use of VHDL fits nicely into the existing design methodology : it allows to use the installed base of VHDL–simulators, and the know–how built up in the past by the designers. As a consequence, the threshold for introducing behavioral synthesis is reduced.

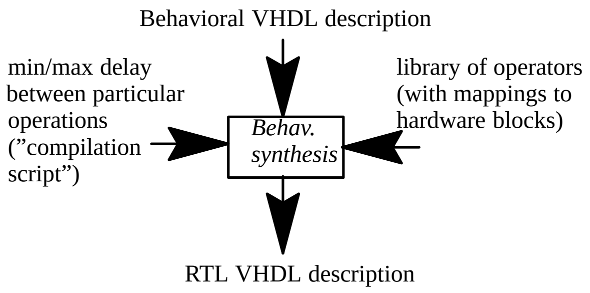
Behavioral VHDL description
min/max delay between particular operations (”compilation script”)
Behav. synthesis
library of operators (with mappings to hardware blocks)
RTL VHDL description

Figure 2 VHDL–based behavioral synthesis.

To be suitable for design–entry, VHDL should offer support for structured programming. This seems to be the case. When writing a C–like description, we only encountered the limitation that in a procedure call, the signal actual must be a static signal : for example, tab(i) can not be passed on as argument, one must use a static signal like tab(0).

In addition, the synthesis tool can impose limitations on the VHDL (cf. logic synthesis tools), e.g. by forbidding the use of multi–dimensional variables as arguments of procedures, which will lead to a less structured design description.

Following table gives an indication of the size of the different VHDL descriptions. Typically we measured that the manual RT–VHDL description size was a factor 2–4 larger than the behavioural one, while the synthesized RT–VHDL was about 5–10 times larger. However, these figures are in some sense misleading, because the description effort to obtain the manual RT–VHDL is much more complicated than the one to write the behavioral VHDL. Some tools provide the RT–VHDL only for examination, not for logic synthesis purposes: the synthesized RT is than far from optimal compared with the manual one.

	# lines VHDL
behavioral VHDL	550
RT–VHDL (manual)	1400
RT–VHDL (synthesized)	3300

It is clear that syntactic invariance does not exist for the current state of behavioral synthesis: the way the designer writes the specification has severe consequences for the result and quality of the final synthesized description. Sometimes, the advantages of the more natural description style at a higher level of abstraction, will completely disappear due to implementation details. The designer has to be aware of the underlying synthesis algortihms!

3.2.2. Validation and performance evaluation

Due to the high data–throughput requirements, the following hardware architecture is typically used to implement typical designs applicable for behavioral synthesis:

- *an application–specific computation unit.* It consists of tuned datapaths, registers and busses or multiplexers to implement the connectivity.
- *a controller for the processor.* Traditionally it is a simple sequencer, that describes the actions during each counter–step. These actions are executed conditionally. These architectures typically use very long instruction words (VLIW).

Validation of the synthesis results can be done with the help of the generated RT–VHDL code, and the information from the synthesis reports. The tools provide usually a state–machine description of the controller, and an overview of the operation schedule across clock cycles, together with the register assignment. A useful feature that is not implemented by many vendors was the computation of the (worst–case) number of clock cycles needed to implement the complete behavioral description, or some statistics about it. Related to this problem, in almost all tools the overhead of loop boundary computations could only be detected by simulating the resulting RT–VHDL code, without any knowledge in the synthesis reports.

In the table below some figures are mentioned in order to give the reader an idea about the quality of existing behavioral synthesis tools. They express the overhead that can be extrapolated out of the experiments we did. In some cases better or worser results can be obtained. These figures are also largely dependent on the tool and on the specific design constraints, and cannot be generalized.

	overhead by synthesis
area (# of gates)	15–20 %
cycle–count (flat code)	10–20 %
timing	10–15 %

3.2.3. Synthesis CPU and Memory usage

Considering the PCC–design, a single iteration through all behavioral synthesis steps takes CPU–efforts from about half an hour to one day, depending on the tool, the constraints and the amount of effort that is asked for doing the job. The experiments were done on a SPARC 5 workstation with 64M RAM.

4. Premises, additional functionality and improved synthesis.

A number of premises for using today's behavioral synthesis tools can be extracted from the design experiences with the available tools. They will be summarized first, and followed by some additional functionalities that are really needed for efficient use of such tools, as a complement to the basic synthesis tasks.

4.1. Synthesis premises

Behavioral synthesis includes some premises, which have to be taken into account before applying it :

- Estimations of low level aspects such as timing and area characteristics are necessary in an early design phase, in order to steer the synthesis and to obtain cost–efficient hardware solutions. This requires access to libraries, containing both predefined and user–defined blocks.
- A practical requirement before applying behavioral synthesis, is the ability to partition the system specification into reasonable chunks of interacting behaviors or processes. Allowing multiple entities, several concurrent processes within an entity, loop hierarchy, and user–defined bit–level arithmetic components can make this partitioning problem feasible, although all tools in our study had limitations in this context.
- An important aspect of behavioral synthesis, is knowledge of the built–in cost–functions of each tool, and the way the user can interactively influence the synthesis results by putting constraints or defining pragma's. The design–community can not live with push–button solutions, but needs tools that allow full exploration of the design space. Not all tools are today in that stage.
- Syntactic invariance seems not realistic for behavioral synthesis. In all tools under investigation for this evaluation the results of behavioral synthesis are heavily dependent on the way the VHDL specification was described. Eliminating such a syntactic variance would ensure that also those designers who have no real expertise in hardware description languages or do not know underlying synthesis algorithms, can obtain satisfactory designs. But, if this will ever occur, behavioral synthesis needs to mature a lot.

4.2. Additional functionality needed for efficient use.

Basically, it is sufficient for the behavioral synthesis tool to generate from an algorithmic description a description that can be read into a logic synthesis back–end tool. However, for real–life use the following additional characteristics are needed :

- The generated solution must be easy to understand, meaning clear and concise RT–VHDL.
- Apart from the RTL–VHDL, the tool should *provide complete reporting*. In case of implemented loop control constructs the designer expects a specific knowledge about the cycle overhead for the exit–continu procedure, in order that he can compute the exact cyclecount of the synthesized design in advance.
- The user must be able to have impact on the solution, without having to modify the behavioral description: specifying constraints and providing key elements of the processor. This is related to the fact that in telecom usually the behavioral VHDL contains a mix of low–level RTL–statements (to describe the detailed behavior of the operations used, e.g. finding the position of the first zero in a bitvector) and high–level constructs (e.g. perform, under certain conditions, a complex operation on certain data).

4.3. Improved synthesis

During our investigation we also encountered more complex issues that intervene sometimes, such as :

- the use of a resource external to the VHDL–entity that is being considered by the behavioral synthesis tool (e.g. single RAM shared by many processes). This is also related to the remaining problem to drive signals in more that one process.
- a behavioral synthesis environment should allow mapping of multiple operators on the same resource, even if the cyclecounts and bitwidths of the operators can differ (e.g. a 8 and 12 bit divisions on the same block).
- the use of (bit–level) user–defined components, to implement specific functions and procedures, is crucial to reduce the computational complexity.
- the need for advanced transformations, such as e.g loop folding, can have a large impact on synthesis results.

- especially for telecom applications, the efficient handling of control flow is extremely important. Indeed, most of the applications are control–oriented.

5. Beyond behavioral synthesis.

When discussing tools that raise the level of abstraction of ASIC design, it is also interesting to identify what these tools do not yet do. The following tasks must currently still be done manually, and are topics for further research, which are necessary for a more global methodology for efficient use of behavioural synthesis.

- Partitioning problem is not solved by behavioral synthesis: e.g. in telecom applications several tasks share the same data in memory. For reasons that the computational complexity of behavioral synthesis limits the size of the descriptions, partitioning, communication, synchronisation and memory sharing has still to be done manually.
- Mixing behavioral synthesis with other tools that go beyond RT–VHDL, e.g. graphical FSM design–entry tools.
- Memory synthesis : At the conceptual level, variables have abstract datatypes. At the design–entry level for behavioral–synthesis, they are placed in RAMs with a specified wordlength and address range. Tasks to be performed are : choice of the optimal datastructure (linked list, hash–table, ...), choice of the number of RAMs and their size,...
- Derivation of complex datapaths : The mapping of multiple related operations on a single datapath (i.e. a calculation unit that is part of the processor) is typically still done manually. Automatic derivation of the structure of a datapath that has different 'modes' is still a topic of research.
- Large–scale design : A more general problem in system synthesis is to start with a behavioral description of the complete ASIC (or board), written e.g. in C++, and to refine, with user interaction, this description until an implementation is obtained consisting of a number of modules that exchange data with each other, use a shared resource, and whose processing is initiated and stopped by a top–level controller. This will lead to heterogeneous systems, with both hardware (behavioral synthesis) and software (e.g. embedded RISC Cores) components. After having integrated behavioral synthesis in the design flow, this can be expected to be the next step in elevating the design entry level.

6. Conclusions.

This paper has described our experiences with VHDL–based behavioral synthesis for non–DSP telecom. The conclusion is that in this area there is distinctly a market potential. Behavioral synthesis leads to reduced design effort complexities, more natural specifications, easier validation and redesign, more exploration possibilities, and less danger for inconsistencies in behavior. Our impressions are based on a number of design experiences with state–of–the–art commercial tools in behavioral synthesis, applied on existing in–house complex telecom applications, and compared with our expertise of traditional design approaches.

However, we indicated that it is not sufficient for the behavioral synthesis tool to simply generate an RTL–synthesizable solution : this solution should be easy to understand (clear RTL–VHDL and complete reporting) and the user must be able to have impact on the solution (specifying constraints and providing key elements of the processors).

It is our belief that commercial behavioral synthesis tools will become general use. For our telecom applications we expect that a potential market of half the size of RTL–synthesis tools will be reached in a few years. However, wide acceptance will depend on improvements of the mentioned shortcomings of todays first generation tools, on the adequate designer trainings, and on clearly defined methodologies for using the tools.

7. Acknowledgments.

The authors would like to acknowledge the interesting discussions with our colleagues at Alcatel–Bell, and with K. Croes of IMEC (Leuven, Belgium).

8. Bibliography.

[1] McFarland M.C., Parker A and Camposano R. "Tutorial on High–Level Synthesis". Proc. Design Automation Conf., June 1988.

[2] Owall V., Andreani P., Brange L., Nilsson P., Wass A. and Torkelson M. "Custom DSP implementation of a GSM speech coder"., Proc EDAC/EuroAsic user forum, pp. 162–165, 1993.

[3] Schultz S. "Behavioral synthesis : concept to silicon". ASIC & EDA, pp. 12–26, Aug. 1994.

[4] Therasse Y, Petit G.H. and Delvaux, M. "VLSI architecture of a SMDS/ ATM router". Annals of telecommunications, 48, nr. 3–4, pp. 166–180. 1993.

[5] Vanhoof B., Bolsens I., De Troch S., Philips L., Vanhoof J. and De Man H. "Design of a voice coder ASIC with the CATHEDRAL II silicon compiler", Proc EDAC/EuroAsic user forum, pp. 150–153.

[6] Vanoostende P., Moerman E. and Van Wauwe, G. "Graphical design–entry above VHDL : myth or reality?" Proc EDAC/EuroAsic user forum, pp. 189–193, 1994.

[7] Handouts 1st Workshop on code generation for embedded processors, Schloss Dagstuhl (D), Sept. 1994.

[8] Gajski D.D., Ramachandran L.,"Introduction to High–Level Synthesis", IEEE Design & Test of Computers, Winter 1994.

[9] Vanoostende P., Van Wauwe G., "VHDL–based behavioral synthesis: can it pay off for telecom ASIC design ? , IFIP Workshop on Logic and Architecture Synthesis, Grenoble, 1994.

[10] IEEE Standard VHDL Language Reference Manual, IEEE, New York, 1988.

[11] Jerraya A.A., Park I., O'Brien K., "AMICAL: an interactive, high–level synthesis environment", Proceedinge EDAC, Paris, France, pp 58–62, Febr. 1993