Efficient Integration of Behavioral Synthesis within Existing Design Flows

W.O. Cesário*, Z. Sugar**, I. Moussa** and A.A. Jerraya*
(*) TIMA Laboratory, 46 av. Félix Viallet - 38000 - Grenoble - France
(**) Arexsys, 1 Chemin du Pré Carré, 38240 - Meylan - France

Abstract

This paper analyzes the reasons why behavioral synthesis was never widely accepted by designers, and then we propose a practical solution to this problem. The main breakthrough of this new approach is the redefinition of the synthesis flow at the behavioral level to better profit from the powerful of RTL and FSM synthesis tools. The effectiveness of this new methodology is illustrated with two large design examples: a 2-million-transistor ATM shaper design and a motion estimator for a video codec (H261 standard).

1. Introduction

In theory, behavioral synthesis brings a set of eagerly wanted advantages: short design development cycles, early estimation of system performance, better design reuse, technology independence, improved area-timing-power-throughput tradeoffs, etc. However, some problems prevented it from being widely used in industry. Among these problems, the most evident are: synthesis results are difficult to predict and to understand, the input model is not practical, it is difficult to integrate behavioral synthesis in existing design flows and, in many cases, RTL tools achieve better results.

This paper targets the specific problem of integrating behavioral and RTL synthesis in an efficient way. Traditionally, behavioral synthesis performs scheduling, allocation and binding to produce a controller/datapath architecture. While the scheduling step is specific to behavioral synthesis, the rest of the flow could be performed by RTL synthesis as well. Figure 1 shows that there could be a functionality overlap (on gray) between behavioral and advanced RTL synthesis tools. As we will demonstrate in the next sections, a better integration of behavioral and RTL synthesis is possible when resource allocation/binding are done by the latter. The advantages of this new approach to behavioral synthesis comes from a unique reason: an RTL model that is easy to understand and efficient to synthesize could be generated just after scheduling.

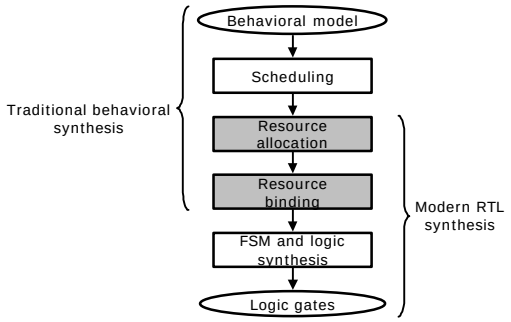


Figure 1 - Scope of synthesis

1.1. Objectives

The main objectives pursued by our new behavioral synthesis methodology are:

- Better integration of behavioral synthesis within existing design flows. It is necessary to seamless integrate modules generated using behavioral synthesis with modules generated by other methods and especially RTL synthesis. This integration is required before and after the execution of behavioral synthesis.
- Generation of efficient design models for RTL synthesis. The coding style generated by behavioral synthesis has to take into account RTL synthesis requirements in order to produce efficient designs in terms of performance, power and gate count.
- Generation of understandable solutions from behavioral synthesis. Even if the code produced by behavioral synthesis is not aimed to be read by designers, they will always require a readable model in order to be able to understand the behavioral synthesis process and to debug the design.

1.2. Structure of the paper

This paper is organized in the following way. First, we show the main limitations of the previous generations of behavioral synthesis tools. Next, the main features of our

scheduling-based behavioral synthesis methodology are presented. Then we present the synthesis results for two large applications. Finally, we discuss the advantages of the new methodology and we draw our conclusions.

2. Behavioral-level design methodologies

Research on behavioral synthesis systems may be classified into three generations of tools. The first generation of behavioral synthesis tools was developed mainly by the computer architecture community [1][2]. The most important result obtained was the precise definition of the synthesis tasks. RTL synthesis was not yet a reality at the time, so each tool had to do module selection, lower level synthesis and even layout generation.

The second generation is characterized by the choice of restricted application domains and the generation of a controller/datapath architecture. Tools concentrated on fewer synthesis tasks (scheduling, allocation and binding) and relied on RTL synthesis for FSM generation and technology mapping. More than hundred behavioral synthesis tools have been reported [3][4] for a variety of application domains: DSP, embedded controllers, communication circuits, etc. The main drawback with these tools was the rigid target architecture, over-structuring of the output lowered the efficiency of RTL synthesis. This is mainly due to the difficulty of doing RTL optimizations over the borders of the structural blocks. However, when the hierarchy is flattened down the model obtained is too low level (gates) to enable any high-level optimizations. Recently, some tools have tried to improve on previous work by allowing a flexible order for the execution of the synthesis tasks [5][6].

The new generation of synthesis tools tries to take full profit from the expertise and industrial investment in RTL synthesis tools. The main breakthrough in these new systems is the adoption of a new design flow. In an early work, VOTAN [7] has tried a re-timing based approach where scheduling was performed as a behavioral VHDL code transformation followed by RTL synthesis. Unfortunately, development stopped before it could be validated in large examples. In our approach, we will follow the same research line using a more powerful scheduling algorithm that enables the generation of efficient RTL code and consequently more optimized results. *Hiasynth* [8] represents a radically different approach to integrate behavioral and RTL synthesis in the same design flow. RTL synthesis is used to execute behavioral synthesis tasks by means of a new design model called "Behavioral Network Graph". *SAW* [9] accepts a "cycle-accurate behavioral code" as input, so it has no restrictions as to where clock edges can be placed. As a result, designers can easily describe arbitrarily complex state machines in a format that is much easier to understand and modify than RTL code. The downside of *SAW* approach is that the whole flow is optimized for designs that are more heavily datapath than control-oriented.

2.1. Contributions

This paper introduces a new generation behavioral synthesis flow. Its main characteristics are:

- The synthesis is centered at the scheduling task, which makes it easier to understand. The output code is a high-level cycle-accurate FSM that allows to fully exploiting the capabilities of RTL synthesis tools.
- The input model is a subset of VHDL that allows mixing cycle-accurate protocols and pure behavioral models within the same specification. The scheduling is able to accommodate several synchronization modes within the same module (VHDL process). This opens the application of behavioral synthesis to a large class of applications where protocol and computation need to be mixed.
- The produced RTL model is easy to understand. The synthesis algorithms make use of a smart naming convention that makes the produced RTL code easy to read by the designer of the behavioral model.

The efficiency of this design flow has already been demonstrated by Moussa [10]. The main result was a three-fold reduction in design effort when using behavioral synthesis and respecting the same design constraints as RTL synthesis with an equivalent circuit area. In this paper, we detail this new flow and compare it with the classical flow in order to analyze the reasons for its success.

3. The new scheduling-based behavioral synthesis methodology

The overall design flow is shown in Figure 2. The design is partitioned into subsystems or modules so that each module can be described and synthesized efficiently. Behavioral synthesis produces RTL code that may be handled with existing synthesis tools. The verification of both the behavioral input model and the resulting RTL code could use the same testbench. This feature is mandatory for practical use of behavioral synthesis. In our case, this was made possible thanks to the input coding style, which allows mixing pure behavioral code and scheduled behavior. It is easy to make the correspondence between the generated RTL code and the behavioral input code thanks to the naming conventions and an on-screen cross-pointing mechanism. This scheme allows behavioral modules to be mixed with RTL modules within the same system design flow.

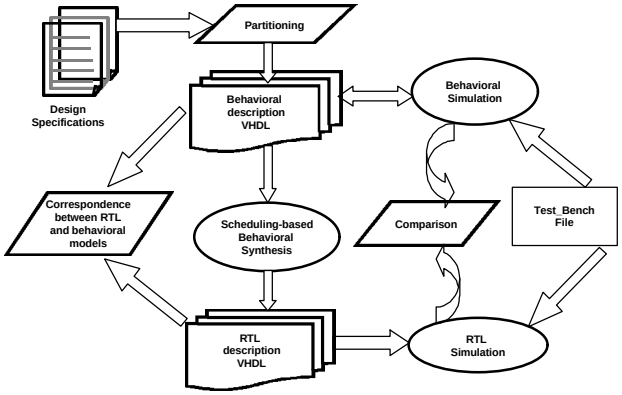


Figure 2 - Behavioral synthesis flow

3.1. Behavioral input model

The behavioral model is an asynchronous VHDL process that communicates with the rest of the system through signals. Synchronization with other modules and processes is performed through *wait* statements. The main facilities provided by our behavioral coding style are:

- *Behavioral wait statement: wait* and *until* expressions may include other signals than the clock. The generated RTL code includes the necessary FSM states and is as efficient as a hand-coded FSM model.
- *Mixing loop, if and wait statements:* this is probably the most significant difference between RTL and behavioral coding styles. Behavioral synthesis could perform several optimizations that are difficult to perform manually (for instance, loop unrolling).
- *Combining wait and procedure calls:* procedures can contain complex control structures and *wait* statements. This feature combined with the above one aid to drastically reduce the size and complexity of the input model. Since RTL synthesis forbids *wait* statements within procedures, all procedure calls need to be in-lined.
- *Description of complex process:* in RT-level, complex processes are split into smaller, more manageable processes. This strategy has the drawback of requiring extra lines of code to manage the inter-process communication. Behavioral code empowers the use of more complex processes reducing the necessity of heavy inter-process communication.

The behavioral specification has two main kinds of operations: synchronization and computation statements. Our model assume that the behavioral description is a system-level FSM where the *system states* correspond to synchronization points and *system transitions* are made of the code executed between two synchronization points during a simulation step. System transitions are also called *execution threads*. These may include sequential statements that form loops, have conditional branches or evaluate expressions and assign them to variables. Execution delays of system transitions are transparent from the point of view of the external environment. It means that the order of operations can be changed during scheduling as long as the order of input/output behavior of the system remains unchanged. Obviously, shifting I/O operations in time is forbidden.

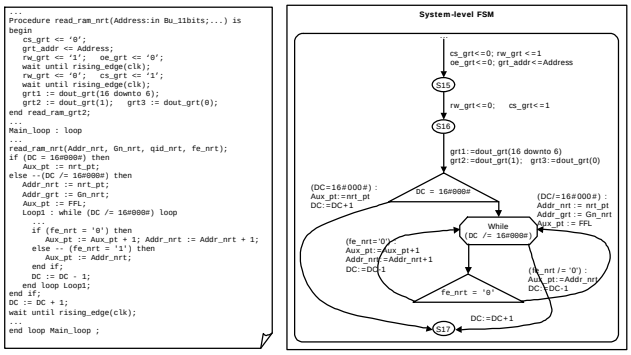


Figure 3 - The VHDL specification (a) and the system-level FSM input model (b)

Figure 3 shows a simple example, it consists of a segment of VHDL description extracted from the ATM Shaper application (Figure 3a) [10] and the corresponding system-level FSM (Figure 3b). The code uses a procedure call containing *wait* statements to access a RAM memory. The system state **S17** corresponds to the *wait* statement in the body of VHDL code. States **S15** and **S16** correspond to the *wait* statements inside the procedure *read_ram_nrt*, they realize the communication protocol with the memory. The system-level FSM (Figure 3b) mixes traditional FSM notation and control-flow graphs (CFG). Different transitions that start from the same state may share the same code allowing for a very compact notation. Obviously, not all transitions can be executed in a single clock cycle and have to be partitioned. For instance, some of the transitions starting from state **S16** include a data-dependent loop that needs to be broken into clock cycles. The next section details how the scheduling of this model produces a cycle-accurate FSM that may be handled using classical RTL synthesis tools.

3.2. Scheduling-based behavioral synthesis

Scheduling consists mainly in partitioning complex system transitions into single clock transitions. Of course, this operation may induce the creation of new states. The scheduling method used here starts from the system-level FSM model and produces a cycle-true FSM. Two steps are used during this process.

The first step deals with the implicit states of the behavioral description. These come mainly from data-dependent and infinite loops that do not include synchronization points. Loops with fixed bounds also introduce extra states it they are not unrolled. This step

makes use of a modified *Dynamic Loop Scheduling* algorithm [12]. It produces a new model where transitions are formed by operations that can be executed in parallel in one control step. Figure 4a shows the FSM generated for the VHDL code presented before. This step assumes that we have an infinite clock period and an infinite amount of resources.

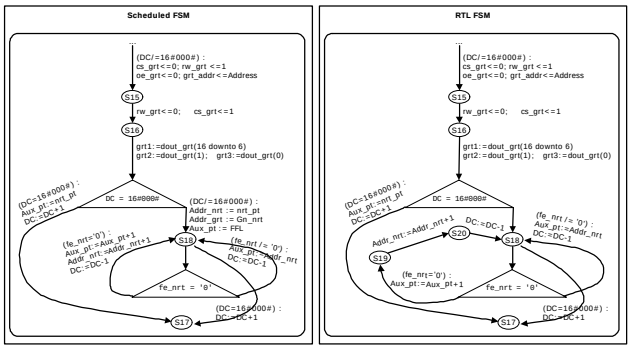


Figure 4 – Scheduled FSM (a) and RTL FSM (b)

In the case of our example, an extra computation state (**S18**) has been inserted due to the data-dependent loop. The initial system transitions have been partitioned into three loop-free transitions. This implies that an execution condition must be computed for each transition (the conditions are represented inside parenthesis in Figure 4).

The second step performs a system-level re-timing in order to ensure that all transition can execute in a single clock cycle. It introduces new states according to implicit and explicit constraints. Implicit constraints are necessary to ensure the correct execution of operations (data dependencies between conditions and expression evaluations). Explicit constraints are imposed by the designer such as primary resource constraints and the clock period. Because scheduled-FSM transitions do not contain loops, classic scheduling techniques are well suited and we apply a LIST scheduling [13] technique. In our example (see Figure 4b) the constraint was to use only one ALU. This resulted in the creation of two additional states **S19** and S**20**. Multicycle operations are supported and may be specified using constraints for binding some operations to specific operators. They are represented as a list of transfers that needs to be executed in a fixed order during a set of consecutive states.

One should note that this scheduling scheme keeps a compact representation style allowing to share code between transitions that starts from the same state. This coding style will also be kept in the produced RTL code, as it will be explained later.

3.3. Generated RTL model

The produced RTL code is derived directly from the RTL FSM. It is readable and easy to understand, except for the portion of the code that mixes loops and *wait* statements. This comes from the fact that the system unrolls the loops and introduces extra intermediate variables in order to perform the chaining of operations. The RTL model uses an efficient coding style for synthesis, its main features are:

- *Coding state machines:* the template used to generate RTL code follows the guidelines for coding state machines [14].
- *Flat representation:* enables full optimization by FSM and logic synthesis avoiding the boundary barriers imposed by unwanted hierarchical structures.
- *Resource allocation/binding are left to RTL synthesis:* FSM and logic synthesis is able to combine aggressive logic optimization with resource allocation and binding. For example, a multiplier by a given constant can be one-fourth the area of a general multiplier, particular specialized multipliers can be even smaller, depending on the constant [15]. Further logic optimizations can be possible between the generated modules and the surrounding logic gates.
- *Condition coding:* the scheduling step keeps the structure of the conditional expressions. This allows RTL synthesis to perform comparator sharing, data chaining and control chaining.
- *Readable RTL model:* the generated code use a naming strategy that keeps as much as possible the names provided by the designers.

Examples of generated RTL code will be shown in section 5.

4. Assessing the new behavioral synthesis flow on large examples

In this section, we compare the results obtained with our behavioral synthesis tool against traditional tools. Two design examples were used: *Asynchronous Transfer Mode* (ATM) traffic shaper and a motion estimator for a video coder/decoder. The main features of the ATM Shaper used here are the following: the bandwidth is 155 Mbits/s supporting all types of traffic such as Variable Bit Rate (VBR), Constant Bit Rate (CBR), Unspecified Bit Rate (UBR) and Available Bit Rate (ABR) [16] and up to 4K connections can be managed simultaneously. The main timing constraint states that the processing of an ATM cell cannot exceed 106 clock cycles at 40 MHz operating frequency. More details about this application may be found in [10]. *H261 motion estimator* (HME) is a circuit that determines the motion vectors for the moving parts of an image according to the H261 video-conferencing standard [17]. The HME computes the “distance” or distortion between the current sub-window and the target sub-window, for a total of 256 possible motion vectors.

4.1. Synthesis results

Table 1 compares the results obtained with our behavioral methodology and traditional tools. For executing traditional behavioral synthesis, we used a public domain tool called AMICAL [11]. We think that the comparison with other high-level synthesis tools will be of the same order. The size of the behavioral input model is about the same but the size of the generated RTL code is about 70% smaller with our new methodology. This is explained by the extra hierarchy of controller-datapath RTL architecture. This over structuring lowers the efficiency of RTL synthesis due to the difficulty of optimization over the borders of the blocks. In terms of gate count, our solutions were 10% better in average but also 20% faster. The simplified synthesis flow used in our new methodology fastens the development cycle.

Design		Behav. VHDL lines	Traditional behavioral synthesis			New behavioral synthesis		
			VHDL RTL lines	Gates	Critical path (ns)	VHDL RTL lines	Gates	Critical path (ns)
HME		290	6021	7850	11.19	538	4200	8.49
Shaper	Sched	781	4176	5060	19.57	1476	4350	13.82
	Sender	621	3267	3930	18.31	1289	3840	16.46
	Abrsa	425	3765	3220	18.76	952	3440	14.35
	T. unit	115	985	730	16.14	209	800	13.25

Table 1 - Behavioral synthesis results

5. Evaluation of the new methodology

5.1. Efficiency of the produced RTL model

The coding style of the generated RTL model follows guidelines for coding state machines [14]. Latches are avoided by initializing all signals and providing default signal values. Registers are inferred by updating all signals at the falling edge of the clock; only negative-edge flip-flops are used. By construction, combinatorial feedback is never generated. The code generator is able to produce complete sensitive lists for the signals referenced in the body of behavioral code. Additionally, the coding style favors optimization tasks during FSM and logic synthesis. As explained in section 3.3, the scheduling step keeps the algorithmic structure of the transitions. This allows maximizing resource sharing when implementing conditions. The use of *if-elsif-else* structures to deal with the conditional expressions enables efficient synthesis and sharing of comparators. We obtained gains of about 50% in area due to this strategy. Additionally, this style allows a full exploration of time-area tradeoffs provided by modern FSM synthesis tools. In fact, the non-separation between control and data operations makes it easier to combine control and data chaining. This feature, very difficult to implement in behavioral synthesis [8], is performed very efficiently by FSM and RTL synthesis.

The RTL module generator plays an important role when executed in parallel with resource allocation and binding. For example, there are eight multiplications by different filter constants on the well-known fifth-order elliptical wave filter benchmark. Most traditional behavioral synthesis systems will use a generic multiplier to perform these multiplications. A good RTL module generator could produce much smaller multipliers specific for each multiplication. For instance, in a 0.8µm CMOS technology, a generic multiplier corresponds to an area of 368 (in generic units) while the largest specific multiplier generated had an area of 295. The differences were between 20% and 38% and represented a 10% gain in total area and delay for the elliptical wave filter example.

5.2. Integration with existing design flows

In traditional behavioral synthesis, there is a problem to mix subsystems described at different abstraction levels. In our new methodology, behavioral modules can be easily mixed with RTL blocks. Traditional behavioral synthesis tools impose very restrictive specification styles. Thus, designing complex applications required an over-decomposition in sub-modules that could be described in a pure style suitable to some specialized behavioral synthesis tools. These restrictions are mainly due to the scheduling step. In fact, most scheduling algorithms impose restrictions on the input description style. The behavioral input style adopted by our behavioral synthesis tool and its scheduling allow describing complex applications in an intuitive way.

```
...
Procedure read_ram_grt2(Address:in Bu_11bits;
      grt1:out Bu_11bits; grt2: out std_logic;
      grt3:out std_logic) is
begin
   cs_grt <= '0';   grt_addr <= Address;
   rw_grt <= '1';   oe_grt <= '0';
   wait until rising_edge(clk);
   rw_grt <= '0';   cs_grt <= '1';
   wait until rising_edge(clk);
   grt1 := dout_grt(16 downto 6);
   grt2 := dout_grt(1);   grt3 := dout_grt(0);
end read_ram_grt2;
...
Main_loop : loop
…
  else -- (FE ='1') Filled position an
                  ...
  while (RI = '0') loop
      Addr_grt := gn_grt;
      read_ram_grt2(Addr_grt,gn_grt,SG,RI);
      gn_grt := gn_grt + 1;
    end loop;
                  ...
  end if;
                ...
end loop main_loop ;
```

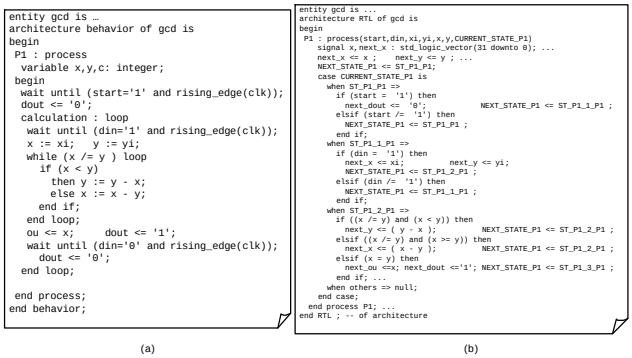
Figure 5 – Mixing behavior and RTL code

For example, Figure 5 shows a sequence including a *loop* and a procedure call that includes two *wait* statements. In fact, the procedure corresponds to a cycle-true communication protocol with a memory block. The full system included several memory blocks described as separated entities. These blocks were described at the clock cycle level with delay annotation. They were necessary for the simulation of the whole system at both

behavioral and clock-cycle level. They were used as black boxes during the synthesis process.

5.3. Understanding the results of behavioral synthesis

Classic behavioral synthesis produces a hierarchical structural model that is very different from the input description. Thus, correlating the input and the output of the synthesis process is nearly impossible. In our case, the output code is easier to understand because synthesis tries to keep the initial control structure. In Figure 6, we could see that all operations and conditions in the behavioral input (Figure 6a) could be immediately identified in the generated code (Figure 6b). The loop structure is broken using an extra state. This kind of transformation may be very hard to perform manually in case of nested loops. During scheduling, several temporary variables may be introduced when splitting states and for some optimization purposes.



```
entity gcd is …
architecture behavior of gcd is
begin
 P1 : process
  variable x,y,c: integer;
 begin
  wait until (start='1' and rising_edge(clk));
  dout <= '0';
  calculation : loop
   wait until (din='1' and rising_edge(clk));
   x := xi;   y := yi;
   while (x /= y ) loop
     if (x < y)
       then y := y - x;
       else x := x - y;
     end if;
   end loop;
   ou <= x;     dout <= '1';
   wait until (din='0' and rising_edge(clk));
     dout <= '0';
  end loop;

 end process;
end behavior;
```

(a)

```
entity gcd is ...
architecture RTL of gcd is
begin
 P1 : process(start,din,xi,yi,x,y,CURRENT_STATE_P1)
   signal x,next_x : std_logic_vector(31 downto 0); ...
   next_x <= x ;    next_y <= y ; ...
   NEXT_STATE_P1 <= ST_P1_P1;
   case CURRENT_STATE_P1 is
     when ST_P1_P1 =>
       if (start = '1') then
         next_dout <= '0';          NEXT_STATE_P1 <= ST_P1_1_P1 ;
       elsif (start /= '1') then
         NEXT_STATE_P1 <= ST_P1_P1 ;
       end if;
     when ST_P1_1_P1 =>
       if (din = '1') then
         next_x <= xi;          next_y <= yi;
         NEXT_STATE_P1 <= ST_P1_2_P1 ;
       elsif (din /= '1') then
         NEXT_STATE_P1 <= ST_P1_1_P1 ;
       end if;
     when ST_P1_2_P1 =>
       if ((x /= y) and (x < y)) then
         next_y <= ( y - x );          NEXT_STATE_P1 <= ST_P1_2_P1 ;
       elsif ((x /= y) and (x >= y)) then
         next_x <= ( x - y );          NEXT_STATE_P1 <= ST_P1_2_P1 ;
       elsif (x = y) then
         next_ou <=x; next_dout <='1'; NEXT_STATE_P1 <= ST_P1_3_P1 ;
       end if; ...
     when others => null;
   end case;
 end process P1; ...
end RTL ; -- of architecture
```

(b)

Figure 6 – Behavioral (a) and generated VHDL (b)

Of course, if more aggressive behavioral-level optimizations (like loop unfolding and chaining) are used the correlation will not be so obvious. Our behavioral synthesis tool has a VHDL correlation mechanism to solve this problem. Every element generated during behavioral synthesis, e.g. a state or variable, has a name that is associated with its original environment in the behavioral specification. So when reading the generated code, the designer has insight from the origin of each name.

6. Conclusion

This paper described a new behavioral synthesis methodology belonging to a new generation of behavioral synthesis tools. The features that distinguish our behavioral synthesis tool from old generation tools are: the close integration with RTL synthesis; abandonment of the controller-datapath architecture as the only possible interface with RTL synthesis; a scheduling approach that leaves the allocation and binding tasks to RTL synthesis and is capable of treating efficiently mixed-style applications; and the generation of efficient RTL code comparable in quality with hand-made code. We demonstrated that this new design flow allows to produce better results than classical behavioral synthesis.

7. Bibliography

[1] M. Barbacci, “Automatic Exploration of the Design Space for register Transfer (RT) Systems”, Ph.D. Thesis, Dept. Of CS, Carnegie-Mellon University, November 1973.

[2] P. Marwedel, “The MIMOLA Design System: Detailed Description of the Software System”, 16th DAC Proceedings, pp. 59-63, New York, USA, June 1979.

[3] D. Gajski, et al., “High-Level Synthesis: Introduction to Chip and System Design”, Kluwer Academic Publishers, Boston, Ma, 1992.

[4] R.A. Walker, R. Camposano, “A Survey of High-Level Synthesis Systems”, Kluwer Academic Publishers, Boston, Ma, 1991.

[5] L. Guerra, M. Potkonjak and J. Rabaey, “A Methodology for Guided Behavioral-Level Optimization”, ACM/IEEE DAC, p.309-314, 1998.

[6] K. Küçükçakar, et al., “Matisse: An Architectural Design Tool for Commodity ICs”, IEEE Design & Test of Computers, April-June, 1998.

[7] N. Wehn, et al., “Scheduling of Behaviorial VHDL by Retiming Techniques”, Proc. of EuroDAC, 1994.

[8] R.A. Bergamaschi, “Behavioral Network Graph: Unifying the Domains of High-Level and Logic Synthesis”, 36th ACM/IEEE DAC, June 1999.

[9] Thomas, Lagnese, Walker, Nestor, Rajan, and Blackburn: Algorithmic and Register Transfer Level Synthesis: The System Architect's Workbench, Kluwer, 1990.

[10] I. Moussa et al., “Comparing RTL and Behavioral Design Methodologies in the Case of a 2M Transistors ATM Shaper”, ACM/IEEE DAC, June 1999.

[11] A.A. Jerraya, H. Ding, P. Kission, M. Rahmouni, “Behavioral Synthesis and Component Reuse with Kluwer Academic Publishers, 1997.

[12] K. O’Brien et al., “DLS: A scheduling algorithm for high-level synthesis in VHDL”, Proc. of the EuroDAC, Paris, France, February, 1993.

[13] T.C. Hu, “Parallel Sequencing and Assembly Line Problems”, Operations Research, pp. 841-848, November, 1961.

[14] M. Keating and P. Bricaud, “Reuse Methodology Manual for System-On-A-Chip Designs”, Kluwer Academic Publishers, 240 pp., June 1998.

[15] A. DeHon and J. Wawrzynek, “Reconfigurable Computing: What, Why, and Implications for Design Automation”, 36th ACM/IEEE DAC, June 21-25, 1999.

[16] The ATM Forum Technical Committee. Traffic management specification v4.0. af-tm-oo56.000 Letter Ballot, April 1996.

[17] A. Wise, “Introduction to Motion picture Coding and the CCITT Algorithm”, December, 1989.