

The Role of VHDL within the TOSCA Hardware/Software Codesign Framework

S. Antoniazzi (a,b), A. Balboni (b), W. Fornaciari (a), D. Sciuto (c)

(a) CEFRIEL, via Emanuelli 15, 20126 Milano (MI), Italy, E-mail: fornacia@mailier.cefriel.it

(b) ITALTEL-SIT, Central Research Labs, CLTE, 20019 Castelletto di Settimo m.se (MI), Italy

(c) Politecnico di Milano, Dip. Elettronica e Inform., P.zza L. Da Vinci 32, Milano, Italy, E-mail: sciuto@elet.polimi.it

Abstract

This paper presents an overview of a hardware/software codesign methodology and its supporting framework (TOSCA) with emphasis on the role played by the VHDL language. The proposed approach is tailored for control-oriented applications and aims at capturing the entire codesign flow, including system cospecification, high-level partitioning, hw/sw tradeoffs, cosynthesis and cosimulation.

VHDL acts as the cornerstone in the task of evaluating the outputs of the codesign flow as well as linking them to commercial simulation and synthesis environments. In particular, the paper will address the issue of simulating hardware/software architectures, focusing on the application of VHDL to the definition of an easily retargetable CPU core model by exploiting the concept of virtual kernel and customizable input/output management scheme.

1. Introduction

The level of interest in balancing the performance of customized hardware with the low cost and flexibility of software components is quickly growing in many application fields such as telecom, automotive and consumer electronics. In order to take full advantage from concurrent design of mixed hw/sw architectures, new design automation strategies providing flexible architectural exploration should be smoothly integrated within standard industrial design environments.

Although hardware/software codesign goals and techniques will not probably converge to a single common interpretation, due to the wide spectrum of application fields and design requirements, the potential value-added provided by the automation of codesign tasks has been shown by a number of recent research works ([1], [2], [3], [4], [5]).

Aim of this paper is to introduce a novel methodology to manage the codesign process for a specific application field, namely control-dominated ASICs such as those embedded into telecom digital switching subsystems. The development of such methodology is currently in progress within a research project called TOSCA (TOols for System Codesign Automation), in which one of the main activities is the definition of a support environment by integrating commercial EDA software with new experimental tools [6].

After a general overview of the codesign methodology, the paper will discuss the main phases allowing cosynthesis and

VHDL-based cosimulation, starting from high-level cospecification. The prototype software environment, supporting the envisaged codesign flow, will also be addressed.

2. The codesign environment

The proposed methodology aims at allowing computer-aided exploration of alternative system architectures in order to balance hardware cost/performance and software flexibility. A prototype software environment, supporting the codesign flow depicted in fig. 1, is currently under development. The final goal is to cover all the following issues:

- acquisition of behavioral specifications, suited to the application field of interest, while maintaining full independence of any particular hw/sw implementation;
- analysis/validation at specification-level;
- tool-directed system-level partitioning including hw/sw binding of specifications;
- concurrent synthesis of sw-bound, hw-bound parts and related interfaces;
- cosimulation to validate the cosynthesis results;
- cost/performance analysis of the alternative hw/sw architectures;
- integration with commercial RTL synthesis and optimization tools, as well as software/firmware development tools.

The codesign process starts from a system model captured via a mixed graphical/textual formalism, based on concurrent and hierarchical finite-state machines (FSMs). After a preliminary analysis/validation activity, an internal system representation (TOSCA DB) tailored to support high-level architectural exploration, is obtained.

The design database is directly involved in the following tasks: the transformation of the initial system modularization to produce a new set of system partitions and their association (*binding*) either with software or with dedicated hardware units; the HW-SW interface generation; the cosynthesis stage; the VHDL-based cosimulation of the resulting architecture.

A set of strategies and basic algorithms can be iterated onto the system representation, until the design constraints are satisfied. The user, as well as some heuristic strategies, can organize these actions along customizable schedules called *recipes*.

Because of the complexity involved by the task of early prediction of the impact of VHDL synthesis and logic

optimization on speed and area for the selected application field (control-oriented systems), the hardware mapping stage does not exploit technology-dependent parameters and classical high-level synthesis approaches to operation scheduling. Instead, we propose an alternative strategy based on initial synchronous specifications and their direct mapping onto synthesizable finite state machines.

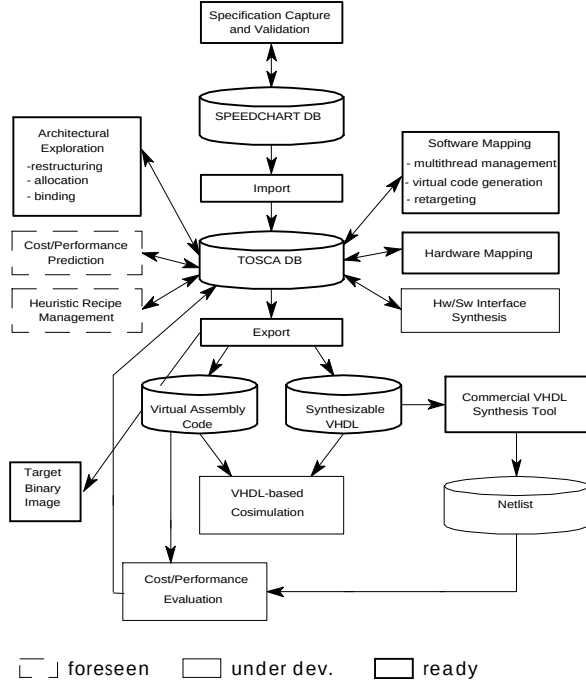


Figure 1: The codesign activities supported by the TOSCA environment.

Due to the requirement of carefully controlling time delay, code size and low level interfacing schemes, the software parts need to be considered at a lower level with respect to C-language based solutions. Our solution is to consider the software description at the level of a virtual assembly instruction set whose structure can be mapped onto different CPU cores with fully predictable translation rules and, consequently, reliable performance estimation.

The final stage in the codesign flow is represented by VHDL-based cosimulation of each proposed hw/sw architecture, in order to achieve feedbacks on the effectiveness of the applied recipes. Moreover, VHDL descriptions provide a suitable vehicle to bring together the codesign flow and the next steps of RTL synthesis and optimization.

4. System design and synthesis

A commercial environment (SPeeDCHART by Speed Electronic) has been adopted for both specification and validation purposes. The formalism provided by SPeeDCHART belongs to the Statecharts family ([7], [8]), coupling graphics with textual descriptions written in a VHDL-like script language. After the Import procedure from SPeeDCHART has

been completed, architectural tradeoffs may be carried out by iterated manipulation of the core model stored within the TOSCA design database. This stage manages two classes of objects (*processes*, namely the modules in the specification, and *architectural units*, i.e. the modules in the target implementation) and involves three tightly related activities:

- *restructuring*: transformations local to a single process (acts at process-level only);
- *allocation*: clustering of processes by assigning an architectural unit identifier to each process (operates on both processes and architectural units);
- *hw/sw binding*: marking of each architectural unit either with a software or a hardware constraint (operates on architectural units only).

In the following, a software-bound architectural unit will be indicated with the term *thread* while a hardware-bound unit will be called *coprocessor*.

Users may define their own custom flows (*recipes*) based upon a set of basic transformation algorithms by means of an interpreted recipe script language (RSL). Recipes may also contain special report actions, describing characteristics and statistics about intermediate and final results. The output of this process is a set of monolithic architectural units with a binding establishing either a hardware or software implementation. Each architectural unit is then passed as input to the subsequent cosynthesis stages.

The target hw/sw architecture is intended to be realized on a single chip. The most general case includes one off-the-shelf CPU core cell and a collection of synthesized *coprocessors*. After restructuring, allocation and binding, each resulting hardware-bound architectural unit is mapped onto its own coprocessor. In this discussion the term coprocessor includes also arithmetic/logic operations and the possible private storage capability, while high-level synthesis tools typically separate controllers from data-paths.

If a coprocessor requires interfacing to/from software-bound elements, then it is connected to the CPU shared data bus (and related address/control lines) and to the interrupt lines. All hardware-to-hardware interfaces are managed by customized local interconnection lines. The RAM memory required for program/data storage shares with coprocessors the main data bus but can be accessed only by the CPU core.

Concerning the *hardware mapping* strategy adopted in TOSCA, it should be pointed out that control-oriented specifications cannot be easily managed by classical high-level synthesis approaches involving operator scheduling.

In fact, circuit speed estimation is very difficult when dealing with descriptions dominated by logic functions, where arithmetic operations are typically restricted to a few sums, subtractions and comparisons (if any of them is present at all). During the next stage involving VHDL translation into a generic netlist, technology mapping and logic implementation, any direct relationship between functional specification and synthesized implementation is lost. Estimating area is also a very hard task. As a consequence, scheduling operators according to estimated propagation delays cannot be considered a realistic approach. In the TOSCA module devoted to hardware

mapping, each hardware-bound architectural unit (possibly obtained from multiple merged processes) is implemented by generating a finite state machine VHDL description. Since the starting point is a synchronous model, no additional scheduling step is needed. The VHDL code generator translates the internal representation of each FSM into a VHDL template compliant to the guidelines for synthesizability enforced by commercial tools such as MGC Autologic and Synopsys. The data flow graphs modeling conditions and actions are translated into VHDL statements included in the related template. The algorithm adopted is able to produce a very readable description by building expressions whenever possible instead of basic assignments for each DFG node. Parameters such as the logic types to be used (e.g. BIT-VECTOR vs IEEE standard packages) can be customized by the user.

In particular cases, such as for instance counters, predefined library components may be preferred to RTL synthesis in order to guarantee an efficient implementation.

The application field requirements have led to discard a C-language based approach for the automated implementation of *software-bound* elements. In fact a high-level language such as C does not allow an accurate control of time delays nor the code size as well as the low level characterization of I/O interfaces. Therefore, a lower level of abstraction has been introduced via the concept of *Virtual Instruction Set* (VIS), comparable with the one provided by a RISC assembly language while maintaining independence from the target CPU core.

The VIS is defined in terms of a register-oriented machine supporting unsigned/signed integer data types (8, 16 and 32 bits) as well as all typical arithmetic/logic operations.

Two types of instruction format are considered:

```

op destination, source
op destination

```

where both *destination* and *source* can be registers or memory references (*source* can also be an immediate operand).

Data transfer from software to hardware and viceversa is modeled via memory-mapped coprocessor registers, associated with each port. The overall code structure, can be viewed as an interrupt service routine (ISR) partitioned into some segments. Each ISR is composed of virtual code concerning transition and, before returning from the interrupt exception (RTE instruction), the memory address needed to manage the next ISR will be loaded in the special address register A0. The initial value of A0 is the address of the ISR associated with the *entry* node.

At present, a code generation prototype tool has been developed supporting a single software-bound FSM (anyway, multiple machines may be collapsed before software synthesis). Such a tool provides register usage optimization and automated packing of single-bit variables.

Work is in progress in order to manage multiple concurrent software threads with a minimum overhead, adopting a static scheduling strategy. As it will be discussed in the next section, VIS code can be directly executed allowing hardware/software cosimulation.

The final target code (assembly language or binary image) will be produced only after the architectural assessment has been completed. Such retargeting task is implemented by first

generating an ASCII representation and then invoking a rule-based program written in PERL (a text processing language for UNIX platforms). A retargeting tool has been implemented for a Motorola 68000 core. The approach can be easily extended to most popular CPU cores.

4. VHDL-based cosimulation

Although a preliminary validation of the initial system-level specification can be performed by exploiting the SPeeDCHART visual environment, an additional simulation step even at the hw/sw architectural level still represents a significant value-added in order to obtain feedbacks on the effectiveness of the selected design-space exploration recipes. Furthermore, already existing components may be excluded from the specification-level (or managed as *black boxes*) and considered during the cosimulation and logic synthesis stages only.

The cosimulation task involves four main entities: the dedicated coprocessors, the programmable core, the software running on the core and the interface logic. An homogeneous simulation environment based on VHDL has been adopted due to the following reasons:

- VHDL methodologies and tools are already available in most design centers;
- in any case, VHDL models have to be generated as input to commercial register-transfer level synthesis tools regarding dedicated coprocessors and interfaces;
- VHDL language features make possible the concise modeling of programmable cores as well as the simulation-oriented representation of the related software;
- existing hardware modules can be easily included in new projects developed by using the proposed hw/sw codesign methodology.

Cosimulation is more critical for the programmable subsystem with respect to the dedicated hardware parts because it requires VHDL models for the selected CPU core cells whose acquisition can be difficult and/or expensive.

Moreover, a conventional CPU core model (as provided, for instance, by third-part developers) is able to run target binary code only. As a consequence, a specific binary code generator has to be developed for each target CPU (or an assembly code generator if an assembler tool is supplied in addition to the VHDL model).

The proposed approach focuses on minimizing the retargeting effort as well as reducing the number of intermediate steps required to obtain an architectural model ready for cosimulation. The underlying concept exploits the characteristics of the virtual instruction set. In fact, VIS code obtained from software mapping is already optimized for the selected target CPU core and can be executed with no *a priori* partitioning of the code/data memory space due to the virtual addressing scheme adopted.

Each target CPU core model is actually composed of:

- a target-independent kernel executing VIS code;
- a customizable target-dependent I/O module, tailored to manage the bus-based interface to/from the dedicated coprocessors.

In such a way, coprocessors are interfaced to the CPU core through the target bus protocol, while code/data memory representation and access are not explicitly handled at bus level but encapsulated within the virtual kernel.

A VHDL view of the entire hw/sw system architecture, adopting a Motorola 68000 as the target CPU core, is depicted in fig. 1.

```
entity system is
port(
  clk: in std_ulogic;
  reset: in std_ulogic;
  .....
);
end;

architecture system_arc of system is
signal a: std_ulogic_vector(22 downto 0);
signal as: std_ulogic;
signal rw: std_ulogic;
signal uds: std_ulogic;
signal lds: std_ulogic;
signal dtack: std_ulogic;
signal d: std_ulogic_vector(31 downto 0);
signal ipl: std_ulogic_vector(2 downto 0);
...
begin
  cpu: m68k generic map(4096,256,8192)
    port map(clk,reset,a,as,rw,uds,lds,dtack,d,ipl);
  c1: coprocessor1 port map(clk,reset,a,as,uds,lds,dtack,d,...);
  ...
  cN: coprocessorN port map(clk,reset,a,as,uds,lds,dtack,d,...);
end;
```

Figure 1: VHDL top-level representation of a mixed hw/sw system.

The CPU and the coprocessors are connected to the system bus. The control lines are modeled by signals *as*, *rw*, *uds*, *lds*, *dtack*; the address bus and the data bus are represented, respectively, by signal *a* and *d*; finally, signal *ipl* models the interrupt request lines. Note that the CPU core model is parameterized by generics in order to specify:

- code space size (e.g. 4 Kbytes);
- data space size (e.g. 256 bytes);
- physical base address for memory-mapped I/O (e.g. 8192).

All coprocessors share a common conceptual structure, as shown in fig. 2.

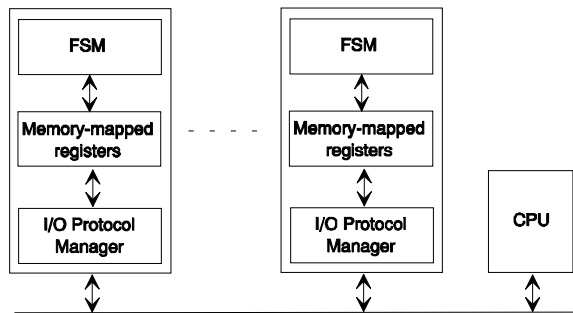


Figure 2: Interface among CPU and coprocessors via bus protocol managers and memory-mapped registers.

Each coprocessing unit is composed of a bus protocol manager, a set of memory-mapped registers and the finite-state

machine as obtained from the previous restructuring, allocation, binding and hardware mapping stages. The protocol manager is synchronized with the CPU clock and it is sensitive to a subset of the configurations on the address bus, each of them selecting a particular entry in the register bank.

The internal structure of the M68000 core model is drawn in fig. 3. The *vkern* module represents the virtual part, performing the fetch/decode/exec loop. The *io_manager* process translates I/O requests from the virtual kernel into the target-specific bus protocol (read and write bus cycles). An abstract representation of the memory space is embedded in the virtual kernel.

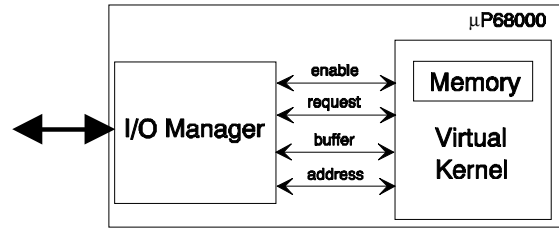


Figure 3: Virtual and target-specific parts in the CPU model.

An overview of the VHDL code implementing the entire CPU entity is presented in fig. 4. The source code follows the modularization depicted in fig. 3. The *io_manager* process communicates with the *vkern* instance through the following signals:

- io_manager_enabled*: used by the *vkern* to enable the *io_manager* during the I/O stages;
- io_manager_request*: specify the kind of *vkern* request (read or write);
- io_buffer*: used for data transfer;
- io_virtual_address*: used by the *vkern* to communicate addresses found in VIS code for translation into physical addresses on the system bus;

The *io_manager* process is composed of two sections modeling the read and the write I/O requests, respectively. Note that bus protocols and transfer speed are strictly dependent on the target CPU. For instance, a complete unidirectional transfer carried out by the M68000 is completed in four clock cycles. As a consequence, the bus managers belonging to the coprocessors have to be synthesized according to such behavioral constraints.

The statement:

```
a <= io_base + io_virtual_address;
```

implements the generation of target physical I/O addresses from virtual offsets.

```
entity m68k is
generic(
  code_size: integer := 1024;
  data_size: integer := 512;
  io_base: std_ulogic_vector(22 downto 0)
);
port(
  clk: in std_ulogic;
  reset: in std_ulogic;
  a: out std_ulogic_vector(22 downto 0);
  as: out std_ulogic;
  rw: out std_ulogic;
```

```

uds: out std_ulogic;
lds: out std_ulogic;
dtack: in std_ulogic;
d: inout std_ulogic_vector(31 downto 0);
ipl: in std_ulogic_vector(2 downto 0)
);
end;
package core_pack is
type io_man_req is (read,write);
type sizes is (byte,word,long);
end;

use work.core_pack.all;
use work.m68k_pack.all;

architecture m68k_arc of m68k is

signal io_manager_enabled: boolean:= false;
signal io_manager_request: io_man_req;
signal io_buffer: std_ulogic_vector(31 downto 0);
signal io_virtual_address: integer;

begin

io_manager: process
begin
if io_manager_enabled then
if io_manager_request = read then
a<="zzzzzzzzzzzzzzzzzzzzzz";
rw<= '1';
wait until clk='1' and clk'last_value = '0' and clk'event;
a <=io_base + io_virtual_address;
wait until clk='0' and clk'last_value = '1' and clk'event;
as <= '0';
uds <= '0';
lds <= '0';
wait until clk='1' and clk'last_value = '0' and clk'event;
wait until clk='0' and clk'last_value = '1' and clk'event;
wait until clk='1' and clk'last_value = '0'
and clk'event and dtack = '0';
wait until clk='1' and clk'last_value = '0' and clk'event;
wait until clk='0' and clk'last_value = '1' and clk'event;
io_buffer <= d;
wait until clk='0' and clk'last_value = '1' and clk'event;
as <= '1';
uds <= '1';
lds <= '1';
a<="zzzzzzzzzzzzzzzzzzzzzz";
else
.....
-- write cycle
.....
end if;
io_manager_enabled <= false;
end if;
end process;

vk: vkernel generic map(8,code_size,data_size,4)
port map(clk,reset,io_buffer,io_manager_enabled,
io_manager_request,io_virtual_address);

end;

```

Figure 4: VHDL code for the CPU entity.

The software mapping stage produces an ASCII file containing a virtual assembly code description. Such file is loaded during the initialization phase of a simulation session into internal data structures. Code and data segments are managed in different ways (fig.5).

```

entity vkernel is
generic(
DBANK_SIZE: integer:= 10;
code_size: integer := 1024;
data_size: integer := 1024;
BUS_READ_DELAY :integer
);

```

```

port(
clk: in std_ulogic;
reset: in std_ulogic;
io_buffer: inout std_ulogic_vector(31 downto 0);
io_manager_enabled: inout boolean;
io_manager_request: out io_man_req;
io_virtual_address: out integer
);
end;

```

use work.core_pack.all;

architecture vkernel_arc of vkernel is

```

type opcodes is (
w_move,
w_and,
w_or,
w_xor,
w_not
);

```

-- CODE MEMORY

```

type kinds is (reg,mem,bv_imm,int_imm,io);
type vis_arg is record
kind: kinds;
bv_value: std_ulogic_vector(31 downto 0);
int_value: integer;
end record;
type vis_instruction is record
opcode: opcodes;
dest: vis_arg;
src: vis_arg;
end record;
type code_memory is array(0 to code_size-1) of vis_instruction;
signal code: code_memory;

```

-- DATA MEMORY

```

type vis_variable is record
value: std_ulogic_vector(31 downto 0);
size: sizes;
end record;
type data_memory is array(0 to data_size-1) of vis_variable;
signal data: data_memory;

```

-- DATA REGISTERS

```

type data_register is std_ulogic_vector(31 downto 0);
type data_bank is array(0 to DBANK_SIZE-1) of data_register;
signal dbr: data_bank;

```

-- CODE POINTERS

```

-- current instruction pointer
ip: integer range 0 to code_size-1;
-- auxiliary pointer for indirect jump
aux_ip: integer range 0 to code_size-1;

```

-- TIMING TABLE

```

type timing_table is array (0 to opcodes'SIZE-1) of integer;
signal t_table: timing_table;

```

Figure 5: VHDL data structures for code/data segments and register bank.

VIS instructions are modeled by the `vis_instruction` record data type containing the opcode (as defined by the enumerative type `opcodes`) and the source/destination operands. Legal kinds for source operands are register (`reg`), memory (`mem`), bit vector/integer immediate (`bv_imm`,`int_imm`) or memory-mapped I/O (`io`), while destination kinds are restricted to register, memory or I/O. Since according to the VIS definition each data transfer has to involve at least one register, direct

transfers from memory to memory (or I/O) are not supported.

The code segment (`code` signal) is implemented as an array of `vis_instruction` records. The current instruction can be referenced through the instruction pointer `ip`. An auxiliary pointer `aux_ip` is used for indirect jumps.

Program variables are implemented via a `vis_variable` record data type whose fields specify content and size (byte, word, long word). The data segment is represented by an array of records.

The virtual kernel also includes a general purpose register bank (`dbr`), whose cardinality is parameterized through the `DBANK_SIZE` generic.

Instruction fetch/decode tasks are reported in fig.6. The fetch operation is implemented by referencing the location in the code array pointed by `ip`. A case construct selects the proper action according to the VIS opcode.

architecture `vkernl_arc` of `vkernl` is

```

...
begin
  process
    variable i: vis_instruction;
  begin
    wait until clk='1' and clk'last_value = '0' and clk'event;
    i := code(ip);
    case i.opcode is

      when jump =>
        ip <= i.src.int_val;
        when ind_jump =>
          ip <= ip_aux;
        when w_move =>
          ...
        when w_and =>
          ...
      end case;
    end process;
  end;

```

Figure 6: Instruction fetch/decode VHDL template.

Such model has to deal with three main issues: the different addressing modes as specified by operand types, the modeling of target-dependent instruction delays by means of a customizable table (`t_table`) mapping opcodes onto the corresponding number of clock cycles and the cooperation between the virtual kernel and the I/O manager in case of operands located in coprocessor memory-mapped registers.

Concerning the last issue, it should be noted that the kernel suspends itself until the I/O manager has completed its own task, by entering into an idle loop as long as some amount of clock cycles (specified by the `BUS_READ_DELAY` parameter) is expired. The delay value related to bus write operations is not needed because it can be computed by subtracting `BUS_READ_DELAY` from the total instruction delay. For instructions not involving input/output, the delay is simply modeled by a waiting cycle activated after the (instantaneous) instruction execution.

5. Conclusions and future developments

In this paper an overview of our approach to hw/sw codesign has been presented. A prototype toolset covering cospecification, hw/sw exploration, cosynthesis and VHDL-based cosimulation have been developed. Work is in progress aiming at introducing more sophisticated algorithms and features on top of such a basic framework.

Currently most of the research effort is devoted to improve exploration recipes and VHDL-based cosimulation. In fact, up to now, restructuring, allocation and hw/sw binding are automatically performed but the choice concerning the strategy to be adopted at each iteration of the exploration cycle is still left to the user. Cosimulation will be improved by increasing the set of supported CPU cores and by providing a complete integration with commercial VHDL compilers and simulators.

References

- [1] Gupta R.K., Coelho C. and De Micheli G., Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components, Proc. of the 29th DAC, June 1992.
- [2] Barros E., Rosenstiel W., Xiong X., Hardware/Software Partitioning with UNITY, Proc. of 2nd Workshop on HW/SW Co-Design, Cambridge, Massachusetts, October, 1993.
- [3] Benner T., Ernst R., Henkel J., Hardware-Software Cosynthesis for Microcontrollers, IEEE Design&Test, Vol.10, No.4, December 1993.
- [4] M.Chiodo, P.Giusto, A.Jurecska, L.Lavagno, H.Hsieh, A.Sangiovanni-Vincentelli, A Formal Specification Model for Hardware/Software Codesign, Proc. of 2nd Workshop on HW/SW Co-Design, Cambridge, Massachusetts, October, 1993.
- [5] Steinhausen U., Camposano R., Gunther H., Ploger P., Theibinger M., Veit H., Vierhaus H.T., Westerholz U., Wilberg J., System-Synthesis using Hardware/Software Codesign, Proc. of 2nd Workshop on HW/SW Co-Design, Cambridge, Massachusetts, October, 1993.
- [6] S.Antoniazzi, A.Balboni, W.Fornaciari, D.Sciuto, HW/SW Codesign for Embedded Telecom Systems, in Proc. of ICCD'94, Cambridge, Massachusetts, US, October, 1994.
- [7] Harel D. et al., STATEMATE: A Working Environment for the Development of Complex Reactive Systems, IEEE Trans. on Software Engineering, Vol.16, No.4, April 1990.
- [8] Narayan S., Vahid F., Gajski D.D., System Specification with the SpecCharts Language, IEEE Design & Test, Vol.9, No.4, December 1992.