A Methodology for Control-Dominated Systems Codesign

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

(a) CEFRIEL, via Emanueli 15, 20126 Milano (MI), Italy, E-mail: fornacia@mailer.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 a methodology and a supporting framework for the design of systems composed of hardware and software modules. The aim is to define an approach, tailored for control-oriented applications, to manage system cospecification, high-level partitioning, hw/sw tradeoffs and cosynthesis. The main goals are always to improve design time and costs by supporting a flexible architectural exploration and to achieve a smooth integration within standard industrial design environments.

Our research effort focuses on fulfilling the goal of linking high-level specifications to efficient and cost-effective hw/sw implementations, by investigating techniques such as synchronous cospecification styles, direct machine code generation as well as exploiting the capability of commercial VHDL synthesizers.

1. Introduction

Heterogeneous hardware/software architectures, for many application fields requiring an ASIC approach, may provide a more effective design solution for some target performance/cost figures with respect to fully dedicated hardware implementations. Therefore, new design automation methodologies should be placed on top of current ASIC design flows in order to integrate dedicated logic obtained by register-transfer level synthesis with CPU core cells and the related software (firmware).

Although hardware/software codesign goals and strategies 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.

The approach considered within the COSYMA project [1] assumes as input of the codesign flow a textual specification written in the C^X language, a C extension supporting task-level concurrency and timing constraints. Such a specification is translated into an internal representation (Extended Syntax Graph) on which preliminary simulation and profiling can be carried out. The environment provides an automatic partitioning stage based on a simulated annealing algorithm. The general strategy assumes an initial fully-software solution and, exploiting information obtained from profiling, carries out hardware extraction (nodes to be moved to hardware are marked in the ESG). Candidate solutions are compared by applying a cost function to the marked ESG. After hw/sw partitioning, hardware-bound parts of the ESG are translated into HardwareC language and implemented via the Olympus high-level synthesis system [2]. On the other hand, C source code is generated from software-bound parts. Hardware/software interfacing is based on template protocols.

A dual approach to automated hw/sw partitioning is presented in [3]. The front-end and the back-end stages are conceptually similar to COSYMA: a textual specification (written in HardwareC) is translated into an internal graph-based representation (System Graph); after hw/sw partitioning, parts targeted to hardware are synthesized by Olympus while C code is produced for software by exploiting a coroutine-based multiprocessing scheme. The main difference can be found in the strategy adopted by the partitioner (Vulcan II). Starting from the System Graph, two sets of graphs (hardware-bound and software-bound, respectively) are generated. In the initial solution most of the design is bounded to dedicated hardware while the only functionalities in the software partition are related to operations characterized by nondeterministic delays (e.g. synchronization primitives and data-dependent loops). An iterative process moves operations between the partitions with the goal of reducing communication overhead while satisfying timing, bus/processor utilization and feasibility constraints. A hw/sw interfacing mechanism is provided, based on polling or FIFO buffering.

Codesign environments not emphasizing the automation of the hw/sw partitioning stage have been also proposed. In [4] system specifications are modeled by asynchronous non-deterministic finite-state machines (CFSM) which, in perspective, will be obtained from a VHDL or ESTEREL front-end. The internal representation of CFSM (SHIFT) is suitable for preliminary analysis by formal verification techniques. Each SHIFT module is manually assigned to a hardware or a software implementation and translated into an equivalent synchronous deterministic FSM. Hardware synthesis is carried out by the SIS tool while software is implemented in terms of a C-language function. Such function is activated by the events associated with the

0-8186-6315-4/94 $04.00

FSM, computes the next state and produces output events. A micro-kernel, customized for each supported micro-controller, provides scheduling for multiple FSMs.

An alternative solution to hw/sw binding is shown in the CASTLE project [5]. Systems are modeled in standard languages such as VHDL, Verilog and C. The internal representation (Software View) is hierarchical and composed of control-flow graphs and basic blocks. High-level transformations such as loop unrolling and lifetime analysis can be also performed. The CASTLE approach, whose implementation is currently under development, is based on the concept of a library of complex components (processors, memories, special-purpose off-the-shelf chips as well as ASICs) and a library-driven mapping strategy. Concerning the hardware synthesis stage, a scheduling algorithm considering resource constraints and minimizing the number of clock steps has been developed, followed by a resource allocation process.

Finally, some results not concerning the entire codesign flow, but focusing on the specific issues of system partitioning and hw/sw binding, have been presented in recent works.

In [6] a methodology is reported based on a formal language (UNITY) and the related theory and proof system. The description style aims at capturing parallel computations in a declarative way using variables to model states and representing transitions through variable modifications. In UNITY there is no concept of control flow and both synchronous and asynchronous behaviors can be specified. The partitioning process starts from a qualitative analysis of the specifications leading to a classification of UNITY basic elements (single enumerate assignments without mutual exclusion) on the basis of predefined criteria such as asynchronicity, synchronicity, mutual exclusion and complexity. An iterative two-stage clustering algorithm is then applied: the former exploits the classification results, the latter is concerned with increasing parallelism. Finally, clusters are allocated to a specified implementation architecture, based on a master-slave scheme, where a RISC processor controls the activation of one or more ASICs.

The PARTIF tool [7] allows the user to explore alternative system-level partitions by manipulating a hierarchical concurrent finite-state model (SOLAR). A primitive set of transformation (moving states, merging states) and decomposition (splitting/cutting macro-states) rules has been defined. At each step of an iterative exploration process, users select a candidate rule. Such rule is automatically checked for feasibility (taking into account costs in terms of state/operation number and design constraints) and, if feasible, it is applied producing a modified system that becomes the new input to the next iteration.

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 activity is the definition of a support environment by integrating commercial EDA software with new experimental tools.

After a general overview of the codesign methodology, the paper will discuss the main phases allowing hw/sw tradeoff exploration and cosynthesis starting from high-level cospecification. The prototype software environment, supporting the envisaged codesign flow, will also be addressed.

2. Overview

The proposed methodology aims at allowing the designer to experiment alternative system designs in order to balance hardware cost and software performance. Such a goal can be achieved through a design flow, able to capture the initial specification avoiding as much as possible any implementation bias but, at the same time, suitable to make possible fast architectural exploration and direct integration within existing commercial synthesis environments.

In order to give effectiveness and validate the proposed approach, a prototype software environment, supporting the codesign flow depicted in fig.1, is currently under development.

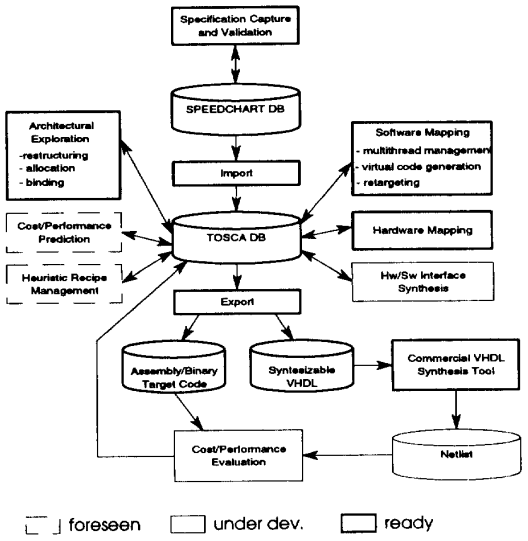

Figure 1: The TOSCA environment.

The target is to cover 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 restructuring and hw/sw binding of specifications;
- concurrent synthesis of sw-bound, hw-bound parts and

related interfaces;
- 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 main activities involving the design database, are the restructuring of the initial system modularization to produce a new set of system partitions and their association (binding) either with software or with dedicated hardware; the hardware-software interface generation; the cosynthesis stage. A set of strategies and basic transformations 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 predefined schedules called recipes.

Both software and hardware synthesis exploit technology-dependent parameters, enabling a realistic cost/performance estimation of each proposed architectural solution.

According to the chosen level of accuracy, the information used for hardware characterization range between purely estimation and data obtained through an actual synthesis process. Due to the impossibility of carefully controlling time delay, code size and low level interfacing schemas, the software parts need to be considered at a lower level with respect to C-language based solutions, in particular if processor retargeting capabilities are envisaged. Our solution is to consider the software description at the level of a virtual assembly whose structure can be mapped onto different CPU cores with fully predictable translation rules and, consequently, reliable performance estimation.

Finally, back-end tools produce a design representation of the selected hw/sw solution (assembly code, VHDL) acceptable for the target commercial implementation environment.

3. System representation

As mentioned in the introduction, the TOSCA approach focuses on control-dominated designs. Therefore, a particular specification style has been adopted, based on a preliminary analysis of the selected target field. Such analysis stage has shown that the following aspects should be carefully taken into account:

a) a concurrent model is required, based on multiple interacting processes;
b) each process may be properly modeled via a synchronous state/transition description;
c) high-level concepts/constructs such as timeouts, behavioral hierarchies and symbolic data may be very valuable in order to cope with specification complexity and implementation independence;
d) logic functions and bit vector data types dominate descriptions while arithmetic data processing plays a secondary role;
e) multiple clocks may drive processes.

It should be pointed out that the synchronous paradigm is not intended to force any hardware bias but to model the intrinsic nature of the application itself. For a discussion of synchronous approaches to reactive system modeling see [8], [9].

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 [10] [11], coupling graphics with textual descriptions written in a VHDL-like script language.

To give a flavor of system specification in the TOSCA/SPeeDCHART environment, fig.2 shows the top-level diagram of a telecom ASIC design; its size, as well as the one of the other examples reported in the following, has been limited only to satisfy the paper length requirement.

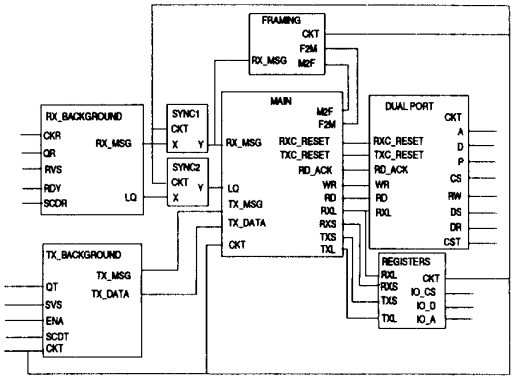

Figure 2: An example of synchronous network.

The system is composed of multiple processes communicating via shared signals. The RX_BACKGROUND process is synchronized by the CKR clock, while all other processes refer to the CKT clock. The two clock signals are unrelated, so that special synchronization elements (SYNC1,SYNC2) have to be introduced.

In fig.3 some of the most significant features of the finite-state specification of a process (MAIN) are shown. The special state labeled *entry* always allows identification of the initial state of a diagram at any hierarchical level. Actions and priorities (represented via *circled numbers* on edges) can be associated with transition edges. State nodes can also have behavioral scripts in terms of entry/exit actions: entry actions (such as the ones associated with the

IDLE state) are triggered when the related state is reached and are semantically equivalent to actions associated with all the incoming transitions; exit actions are triggered when the related state is exited and are semantically equivalent to actions associated with all the outcoming transitions.

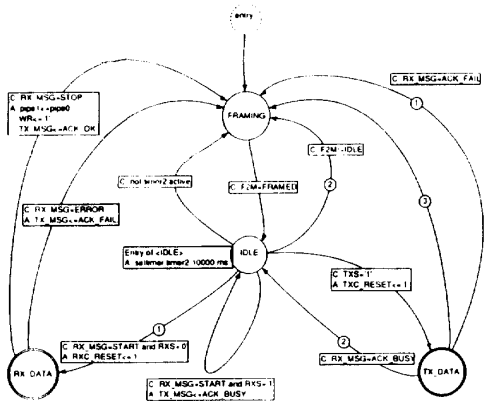

Figure 3: The MAIN process.

The example also shows additional features such as timeouts (captured by the *settimer* statement and the *active* attribute) and the symbolic representation of communication messages (e.g. ACK_OK, ACK_FAIL,...). Furthermore, two hierarchical states (RX_DATA and TX_DATA) are included. Each state is associated with a child diagram (fig.4 zooms on TX_DATA child). Because of the child diagram models a non-terminating process, the only way to stop its activity is to trigger a transition outcoming from the parent state (child overriding).

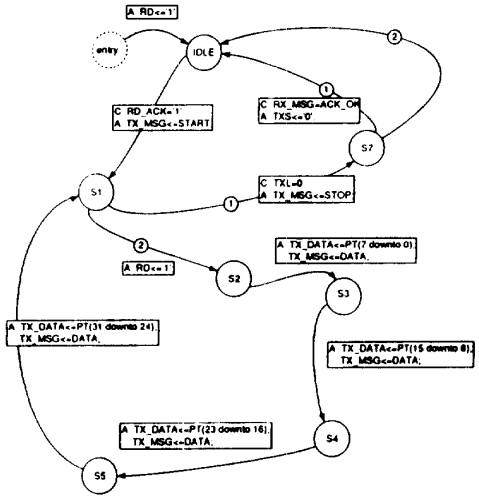

Figure 4: The TX_DATA child diagram.

Finally, the SPeeDCHART environment also allows system validation by simulation, including visual animation capabilities.

Since the subsequent codesign stages have been implemented on top of a different software environment (the O2 object-oriented DBMS) a preliminary step (Import) has to be performed in order to translate SPeeDCHART design data into the TOSCA internal model. Such core representation preserves the concurrent/hierarchical structure of the original specification. The main difference is in the management of the textual scripts: the Import procedure starts from ASCII files produced by SPeeDCHART and carries out a parsing process building symbol tables for declarations and data flow graphs for conditions and actions. Enumerative types are also encoded in this step.

4. System design

After the Import procedure has been completed, architectural tradeoffs may be carried out by iterated manipulation of the core model. 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 modifying the cardinality of the whole process set (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*.

Concerning restructuring, some basic transformation algorithms have been made available:

a) unfolding entry/exit actions;
b) unfolding FSM enabling conditions;
c) flatten hierarchies;
d) mapping timers onto counters;
e) collapsing a set of processes;

As mentioned in section 3, entry/exit actions may be possibly associated with states, in order to allow a more concise representation. Users may preserve such a model structure or apply the algorithm a), obtaining the representation depicted in fig.5, by migrating entry and exit actions respectively toward incoming and outcoming transitions.

SPeeDCHART also allows to specification of a general condition associated with a whole FSM: the FSM is enabled if and only if such a condition is true. The transformation b) moves the condition from the FSM level

downto each transition: such condition is merged with the original transition predicate by using an AND operator.

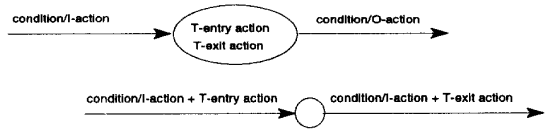

Figure 5: State actions unfolding.

In order to make easier the hardware/software mapping step, hierarchical descriptions can be expanded by applying algorithm c). Referring to the case reported in fig.6, all the incoming edges will be linked to the *entry* state together with their condition/actions pairs. Concerning the outcoming arcs, if no *exit* condition is present, it is sufficient to create an edge connecting the *exit* state with the target state at the upper level. In the general case of edges having their own target state and condition/actions pairs, for each state belonging to the child it is necessary to introduce an edge connected to the upper level state. Actions can still remain joined to the original condition or added to the target state action list.

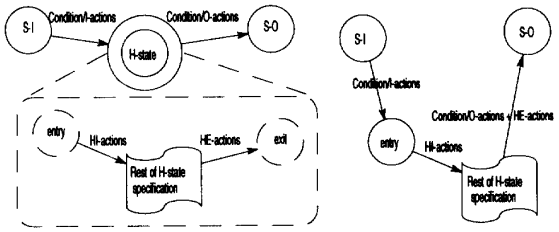

Figure 6: The simplest case of hierarchy removing.

Zero-flagged counter variables with decrement control, introduced by transformation d), are employed to act as special objects modelling timeout conditions.

Finally, the task of collapsing multiple machines on a common final target machine (e) is implemented by a technique based on symbolic execution. A similar approach, but assuming an asynchronous semantics, is discussed in [12]. This capability can be used to collapse machines belonging to a common partition (architectural unit) to obtain a coarse grain description useful for the subsequent hw/sw binding and cosynthesis. The pseudo-code algorithm is reported in fig.6, it is based upon a basic procedure, MERGE, able to collapse two FSMs at once.

Initialization: each initial process is assigned to a architectural unit; each initial architectural unit must be hw or sw bound by the user

For each architectural unit A belonging to the project **do**
CollapsedMachine := first module of A
While a module exist in A **do**
selected := extract module from *A*
CollapsedMachine := **MERGE** (CollapsedMachine, selected)
enwhile
endfor /* a set of collapsed architectural units is obtained */

Figure 7: Structure of the algorithm for building a set of architectural units by pairwise merging of processes.

First actions of the MERGE(MA, MB) procedure are the creation of the interface specification for the merged machine and the removal of the unnecessary inter-module links by transforming them into internal variables. MA and MB, beginning from their first state, are then symbolically executed in parallel. States in the target machine are obtained from each explored state pair (MA,MB). Their DFGs (if any entry/exit action is present) correspond to the merging of those from MA and MB states. The transitions of the merged-machine are obtained by considering all possible combinations of those present in MA and MB. For each resulting transition the following rules are applied:

- actions are merged;
- conditions are combined by an AND operator;
- priorities are managed by computing a function of the original priorities.

Concerning the edges outcoming from collapsed states, the evaluation of the new associated priorities is shown by the simple example reported in fig.8. Each of the initial states have a pair of exiting arcs with their own condition and priority (either 1 or 2). The joined-state outcoming arcs, will have a priority level that is the sum of those associated with the original ones.

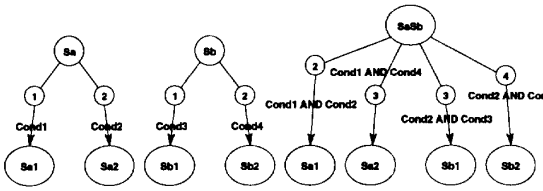

Figure 8 Determination of priorities and conditions associated with the new arcs.

Users may define their own custom flows (*recipes*) based upon the above kit of basic transformation algorithms. 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, as described in section 5.

Additional transformation algorithms are under development, such as moving arithmetic operators across clock steps and splitting a process into multiple subprocesses.

As an example of system restructuring and allocation

through application of the merging and flattening algorithms, let us consider the system composed of three basic machines drawn in fig.9. Two of them, Cell1 and Cell2, have one level of hierarchy whose diagrams are respectively reported in fig.10, fig.11, fig.12 and fig.13; in fig.14 is depicted the only non-hierarchical machine.

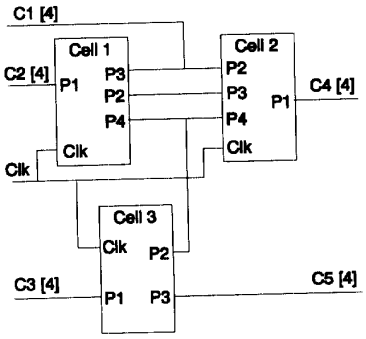

Figure 9: Top view of the entire system.

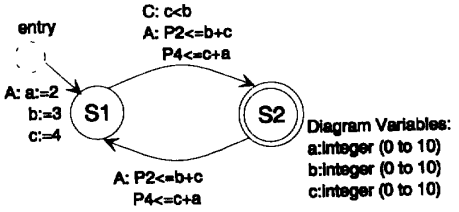

Figure 10: Top level specification of the FSM represented by Cell1 in fig.9.

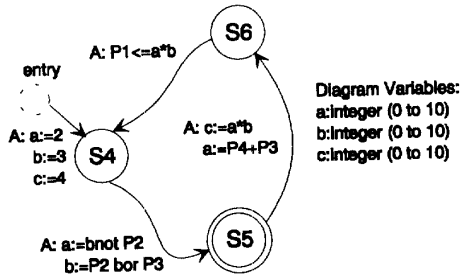

Figure 11: Top level specification of the FSM represented by Cell2 in fig.9.

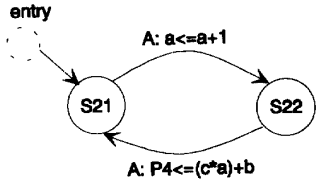

Figure 12 Explosion of the hierarchical state S2 of Cell1.

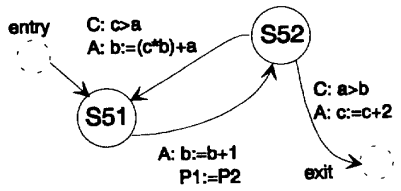

Figure 13: Explosion of hierarchical state S5 of Cell2.

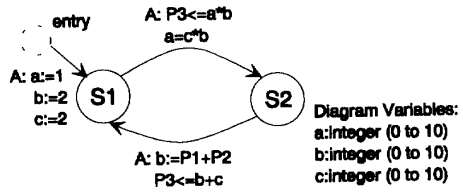

Figure 14: Specification of the FSM represented by Cell3 in fig.9.

The restructuring/allocation process performed onto the initial specification, produces two architectural units (Pα={Cell1, Cell3}, Pβ={Cell2}) and a reduction in the final number of FSMs due to Cell1, Cell3 merging.

The first step has been hierarchy flattening, due to space limit and for the sake of readability, only the state diagram without actions and conditions of the architectural unit Pβ, is reported (fig.15).

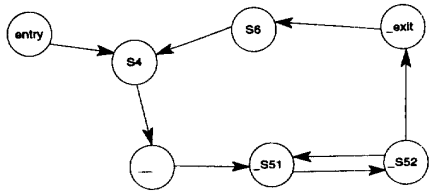

Figure 15 Flattening of Cell2 (architectural unit Pβ).

The description obtained by merging Cell1 and Cell3 after hierarchy removing, i.e. Pα, is reported in fig.16 (for compactness, conditions and actions are omitted).

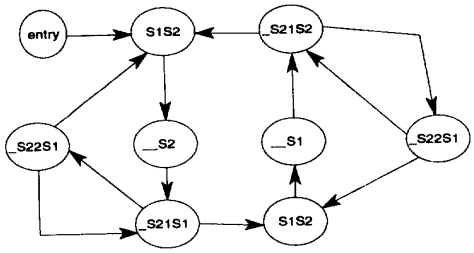

Figure 16 State diagram of architectural unit Pα.

The detailed specification of both actions and conditions, due to space limit, have not been reported.

5. Cosynthesis of hw and sw modules

The target hw/sw architecture is 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 a classical high-level synthesis approach 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 additions, 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 complying 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) or modeling style (structural vs behavioral) 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.

Referring to the example considered in section 4 and under the assumption of hardware binding for both the architectural units, the produced VHDL implementation is represented by two block-encapsulated processes.

The application field requirements have led to discard a C-language based approach for the automated implementation of *software-bound* elements. In fact an 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 the I/O interface. Therefore, a lower level of abstraction has been considered 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 formats 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, accessible via the VIS instruction `write (port,data)`. Moreover, for both *in* and *inout* ports, a RAM variable is also allocated. The refresh of such a variable is performed through the VIS instruction *update*, by considering the information stored in the memory-mapped register *transfer_status*, specifying where input data have to be read. The instruction *update* is constituted by the two following operations:

1) get *transfer_status* value;
2) for each bit = 1 of *transfer_status*, copy the value stored in the register associated to such a bit, in the corresponding RAM variable.

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 between non *drop-thru* nodes and, before returning from the interrupt exception (RTE instruction), the memory address needed to manage the next ISR, will be loaded in register A0. The initial value of A0 is the address of the ISR associated with the *entry* node. A general model of the virtual code is shown in fig.17.

```
Int_Handler:Jump (A0)
S0:
   update
   condition1
   cjump L1
   action 1
   jump L2
   L1:
   .........
   Lm: cond n
   cjump Ln
   action n
```

```
    Lm: write (port, data)
    A0 <- S1
    rte
S1:
    update
    condition1
    cjump L1
    action 1
    jump L2
    L1:
    .........
    Lm: cond n
    cjump Ln
    action n
    Lm: write (port, data)
    A0 <- S2
rte
```

Figure 17: Virtual code structure (only two states are reported).

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. In fig.18, a simple example of register optimization obtained by invoking the tool on a transition condition, is reported. Note that the intermediate results produced by the evaluation process of subexpressions are always kept in machine registers, avoiding expensive memory references.

```
Condition: (((x=y) OR dgd) AND (dj+sks)) > dff
   VIS Code:
   load.w R0, y
   eq.w R0, x
   load.w R1, dgd
   or.w R1, R0
   load.w R0, sks
   add.w R0, dj
   and.w R0, R1
   load.w R1, dff
   gt.w R1, R0
```

Figure 18: Translation of a condition into the corresponding VIS code.

Work is in progress in order to manage multiple concurrent software threads with a minimum overhead, adopting a static scheduling strategy. VIS code is finally translated into the target assembly language (or binary image) by first generating an ASCII representation and invoking a rule-based program implemented by means of a PERL (a text processing language for UNIX platforms) script. A retargeting tool has been implemented for a Motorola 68000 core. The approach can be easily extended to most popular CPU cores.

6. Conclusions and future developments

This paper has presented a methodology suitable to support hw/sw codesign. A prototype toolset covering cospecification, hw/sw exploration and cosynthesis has 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 implementation effort is devoted to the transformation algorithms and to the cost/performance evaluation, while restructuring and hw/sw binding can be performed only manually (the choice concerning the strategy to be adopted at each iteration of the exploration cycle is left to the user). Research is in progress to introduce a higher level of automation, in particular the following criteria are envisaged to drive such activities: direct user selection; clustering according to common logical clock rates; analysis of data flow connectivity; analysis of resource (arithmetic, memory,...) complexity and optimization. Global cosimulation is one of the issues that will be addressed in the future work; both the specification level and the implementation levels will be developed.

References

[1] Benner T, Ernst R., Henkel J., Hardware-Software Cosynthesis for Microcontrollers, IEEE Design&Test, Vol.10, No.4, December 1993.

[2] G. De Micheli et al., The Olympus Synthesis System, IEEE Design and Test of Computers, Vol.7, N°5, October 1990, pp.37-53.

[3] Gupta R.K., and De Micheli G., Hardware-Software Cosynthesis for Digital Systems, IEEE Design&Test, September 1993.

[4] M.Chiodo, P.Giusto, A.Jurecska, L.Lavagno, H.Hsieh, A.Sangiovanni-Vincentelli, Synthesis of Mixed Software-Hardware Implementations from CFSM Specifications, Proc. of 2nd Workshop on HW/SW Co-Design, Cambridge, Massachussetts, October 1993.

[5] Steinhausen U., Camposano R. et alii, System-Synthesis using Hardware/Software Codesign, Proc. of 2nd Workshop on HW/SW Co-Design, Cambridge, Massachussetts, October, 1993.

[6] E.Barros, W.Rosenstiel, A Method for Hardware Software Partitioning, in Proc. of Compeuro, IEEE CS Press, 1992, pp.194-199.

[7] Ismail T., O'Brien K., Jerraya A., Interactive System-level Partitioning with PARTIF, Proc. of EDAC'94, Paris, France, February, 1994.

[8] Benveniste A. and Berry G., The Synchronous Approach to Reactive and Real-Time Systems, in Proc. of the IEEE, Vo.79, No.9, September 1991.

[10] 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.

[11] Narayan S., Vahid F., Gajski D.D., System Specification with the SpecCharts Language, IEEE Design & Test, Vol.9, No.4, December 1992.

[12] Wolf W., Takach A., Huang C., Manno R., The Princeton University Behavioral Synthesis System, 29th DAC, 1992.