Co-synthesis and Co-simulation of Control-Dominated Embedded Systems

Alessandro Balboni (a), William Fornaciari (b, c), Donatella Sciuto (b)
(a) ITALTEL-SIT, Central Research Labs, CLTE, 20019 Castelletto di Settimo m.se (MI), Italy
(b) Politecnico di Milano, Dip. Elettronica e Informazione, P.zza L. Da Vinci 32, Milano, Italy, E-mail: sciuto@elet.polimi.it
(c) CEFRIEL, via Emanueli 15, 20126 Milano (MI), Italy, E-mail: fornacia@mailer.cefriel.it, fornacia@elet.polimi.it

Abstract

This paper presents a methodology for hardware/software co-design with particular emphasis on the problems related to the concurrent simulation and synthesis of hardware and software parts of the overall system. The proposed approach aims at overcoming the problem of having two separate simulation environments by defining a VHDL-based modeling strategy for software execution, thus enabling the simulation of hardware and software modules within the same VHDL-based CAD framework. The proposed methodology is oriented towards the application field of control-dominated embedded systems implemented onto a single chip.

1. Introduction

Many application areas, spanning from telecom systems, automotive equipment to consumer electronics, aims at tradeoff implementation costs and performance requirements, through the use of heterogeneous hardware/software architectures. In particular, for many application fields requiring an ASIC approach to the design, the interest of the CAD developers in methodologies and support tools balancing the performance of customized hardware with the low cost and flexibility of software components has been steadily increasing in the past few years. Therefore, new design automation methodologies should complete current ASIC design flows in order to integrate dedicated logic obtained from behavioral (or Register-Transfer Level) synthesis with programmable parts.

A wide class of computing systems requiring such a detailed design of both hardware and software subparts, fall in the category of *embedded systems*. In the following we will refer to the class of embedded systems which are dedicated to a specific application. For this term we adopt the most restrictive view [Wol94] of a system based on a programmable instruction-set processor which executes a fixed software program, connected to hardware dedicated modules which interact with the external environment and with the processor. They are usually characterized by a *reactive* behavior to the environment stimuli, also conditioned by *real-time* constraints, sometimes requiring a high-performance non-conventional design of the software. Although a certain flexibility/modifiability of the system is always a desirable property, the significant level of customization required by the application, the absence of notable interactions with users and the physical nature of the final product (e.g. a digital switching system, a remote automotive fuel injection controller, small-size portable equipment,...) lead to an almost impossible on-line after production tuning of the system. These reasons, in addition to the time-to-market pressure on delivering as soon as possible the final product, force the designers to freeze the system-level design once the refinement of the initial specification becomes synthesizable. As a consequence, the space for trade-off analysis of different architectural solutions is dramatically shrunk. The result is that a traditional design flow is usually far from a quantitative approach, it is basically driven by a minimum design effort strategy, generating problems during the final modules integration, due to the lack of a concurrent design strategy. The presence of programmable components acts both as a leverage for reducing costs and as a bare support for patching the missing system functionality, since the impact of software modifications is less expensive in comparison with hardware redesign.

However, improving the design process is not only a matter of modifying the designer's view of system-level analysis, the activity of combining mixed hw/sw systems is fairly complex since there is a number of preliminary problems and positions to be carefully considered to successfully obtain an effective overall system synthesis. In fact, to take full advantage from concurrent design of mixed hw/sw architectures, new design automation strategies should be designed to allow a smooth integration with available industrial design environments and standards (e.g. VHDL). One of the challenges/requirements is to achieve *cooperation* instead of *competition* with the EDA (Electronic Design Automation) tools composing the ESDA (Electronic System Design Automation) *arena*.

A basic path for the concurrent design process (*co-design*) starts from capturing the functionality of the whole system (*co-specification*), possibly avoiding any implementation bias. Such a model constitutes the basis for the *system-level exploration,* i.e. the activity of experimenting/evaluating different architectural solutions that will produce two sets of subsystems to be implemented one as hardware and the other as software modules (*partitioning* and *binding*). The final stage consists of the actual synthesis of both the hardware-bound and programmable parts (*co-synthesis*). Some important aspects to be considered along the co-design process are the early prediction of the final results, the possibility of performing a global analysis formally or by simulation (*co-simulation*), the managing and evaluation of different design alternatives, and the reusability support.

Although hardware/software co-design 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 co-design tasks has been shown by a number of recent research works. The proposed approaches can be roughly partitioned in strategies starting from a fully software system implementation moving pieces of software toward the hardware domain and, viceversa, strategies aiming at obtaining the minimum cost by replacing pieces of hardware with software code. Two pioneer researches, representing this duality of goals, are COSYMA [Ben93] and VULCAN-II [Gup92] [Gup93].

The first assumes as input of the co-design flow a textual specification written in the C^x language, a C extension supporting task-level concurrence 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. Candidate solutions are compared by applying a cost function to the marked graph. After hw/sw partitioning, hardware-bound parts of the ESG are translated into HardwareC language and implemented via the Olympus high-level synthesis system [DeM90], while C source code is generated from software-bound parts.

The front-end and back-end stages of VULCAN are conceptually similar to the ones provided by 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, based on iterative process moving of operations between the partitions with the goal of reducing communication overhead while satisfying timing, bus/processor utilization and feasibility constraints.
In the SpecSyn system-design environment [Gaj94a], specifications are captured via the SpecCharts visual language and translated into an internal representation called PSM. Partitioning is performed through algorithms based on clustering or simulated annealing and results are evaluated by estimation tools providing metrics for software and hardware speed, silicon area and code size. An approach to interface synthesis is discussed in [Gaj94b]. A discussion on the importance of having a suitable intermediate format to represent system-level specifications can be found in [Vah95]. The PARTIF tool ([Ism94]) 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.
An alternative solution to hw/sw binding is shown in the CASTLE project [Stei93]. 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. The CASTLE approach 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.
In [Cho94] is presented a co-design environment, called Chinook, tailored to implement reactive real-time controllers with particular emphasis on the problems of modules interfacing and synchronization. The system goal is to cover the problems of partitioning, device synthesis, low level scheduling, code generation and performance estimation. The top-level system specification is captured via a Verilog description while the output consists of all the elements necessary to build the embedded system: the netlist of coprocessor and glue logic together with the assembly code retargeted to the specific microprocessor.
Finally, a co-design environment not emphasizing the automation of the hw/sw partitioning stage based on an extension of the well know FSM paradigm, has been proposed in [Chi93a] [Chi94]. The 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.
The research area concerning hardware/software co-simulation has been widely explored for DSP-oriented applications only [Buc94] [Sut93]. A survey of alternative strategies for more general applications is presented in [Alt91] and [Vah95b] while [Wol94] provides an extensive survey of the existing open research issues and project on embedded system co-design. A specific approach is

proposed in [Gup92] through the Poseidon system, an event-driven scheduler able to manage mixed hw/sw models. Hardware models are simulated at gate-level (i.e. after synthesis and optimization) by the Mercury logic simulator, while the generated software in C language is first compiled into assembly code and managed by a target-specific assembly code executor. In fact, Poseidon acts as a higher-level integrator of multiple heterogeneous simulators.

A different approach showing the different points of view on co-design related issues, can be found in [Wol92]. This research work tends to unify embedded system co-design and distributed systems design within a common conceptual framework.

However, despite the number of proposals appeared in literature, usually they are still too advanced for being considered in current industrial environments. In particular, these approaches do not consider in a unified way the activities of co-simulation and co-synthesis under the point of view of obtaining system representations compatible with current industrial standards. Main purpose of our research is to define a pragmatic approach that could be accepted as prototype in the R&D department of an Italian telecom company. Therefore main requirement is the possibility of integration within an industrial design flow and to allow human intervention on all decisions pertaining system exploration, with the possibility of backtracking along the design process and of reusing already designed submodules.

The paper introduces a novel methodology to manage the co-design process for a specific application field, i.e. control-dominated ASICs, such as those embedded into telecom digital switching subsystems. Usually this class of systems is composed of interacting hardware and software components where a mix of algorithm and event-driven control/communication functions (e.g. protocol stacks) are affected by real-time constraints and have to be captured/processed in an integrated manner.

The development of such a methodology is currently in progress within a research project called TOSCA (TOols for System Co-design Automation), in which one of the main activities is the definition of a support environment by integrating commercial EDA software with new experimental tools [Ant94a] [Ant94b] [Bal95]. One of the main characteristics of the TOSCA framework over other literature proposals is the high level of integration with the existing commercial EDA tools through the direct interfacing to an existing design entry environment (e.g. speedChart), a VHDL representation of the hardware-bound parts, the direct synthesis of the software modules and the possibility of achieving co-simulation of the entire mixed hw/sw system within the same VHDL-based environment. Moreover, the CAD environment allows the user to cover the whole design process, ranging from the system-level specification capture down to the synthesis, by considering low level effects of the software timing properties (at the assembly level), by providing the code for software processes as well as the necessary operating system support and, finally, by generating the interfaces among hw and sw modules. In addition, the internal representation paradigm, based on process algebra [Hoa78] [Jpb94], provides the possibility of extending the analysis to include formal verification capabilities.

The two main phases of the TOSCA co-design environment discussed in this paper are the hardware and software co-synthesis and the co-simulation of the system obtained. Therefore the paper is organized as follows: section two gives an overview of the TOSCA design flow and defines the objectives of the supported co-design activities. Then section three introduces the target system architecture and details the synthesis of software and hardware modules, it also discuss the problems related with direct synthesis of interfaces and software retargeting. An example of software synthesis is reported in Appendix 1. Section four introduces the co-simulation engine while showing the strategies adopted to simulate both hardware and software modules by using the same VHDL-based environment.

The final section outlines the main results of our approach and points out the future research guidelines.

2. The TOSCA design flow

System design companies need methodologies and tools suitable to be integrated within existing design flows, usable by designers with a low training effort and able to move down from abstract specification toward the actual implementation, while maintaining the capability of performing an efficient global co-simulation at any considered level. The *state of the art* of research in this field, tends to focus on specific aspects of the problem or to be affected by an high level of customization, making difficult a significant cooperation with existing design environments.

The methodology we are proposing aims at providing a complete stand-alone design framework, whose main structure is shown in fig.1, allowing a realistic integration with commercial EDA tools as back-end or front-end to the global framework.

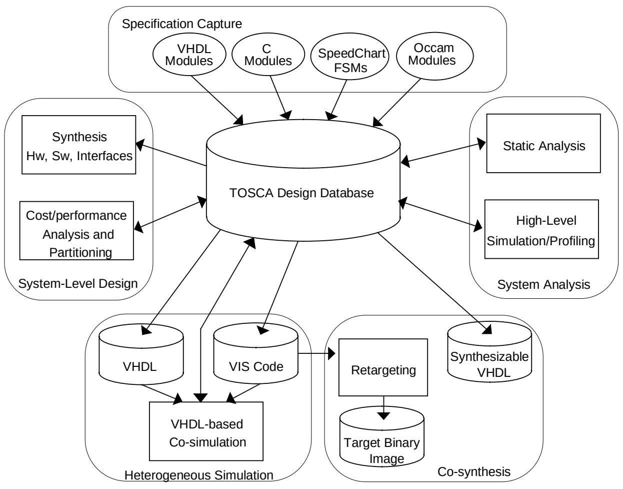


Figure 1. The architecture of the TOSCA co-design environment.

The analysis of the basic design flow followed by system designers has allowed us to identify the proposed automated methodology which takes into account the needs and requirements of the designer. The co-design process supported by the TOSCA CAD environment is composed of the following, partially interleaved, coarse-grain phases:

1. *System description*, including functional requirements, performance goals and feasibility constraints.
2. *Analysis of the system specification* through a set of metrics for early prediction of the implementation results.

3. *System-level partitioning and binding,* to identify a decomposition in modules of the initial specification together with their implementation technology. Manipulations of the system specification, preserving its functionality, are performed. They are driven by the set of metrics which estimate the final results and by the design constraints.
4. *System-synthesis,* producing the assembly code for the software part and VHDL for the hardware-bound modules.
5. *Co-simulation* of the final system and user-level *back-annotation* of the results.

The acquisition of the specifications requires the possibility of integrating elements obtained from different sources, since often most designs are realized starting from previous product releases or make use of existing, or third party, modules. Therefore, the value added of a co-design methodology is tightly conditioned by the achievement of a realistic integration with current trends of specification standards. The characteristics of most project specifications, which are not unconstrained, can be summarized as follows:

- Some parts can be *a priori* hardware or software bound; this can derive from direct intervention of experienced designers or forced by top-level specifications/requirements.
- Some parts can be not only hardware or software bound but already synthesized, e.g. because of reusability of proprietary (or widely accepted) software algorithms and hardware subsystems (e.g. belonging to the company library of VHDL models).
- Design centers with up to date CAD environments can produce models captured via a mixed graphical/textual formalism, based on concurrent and hierarchical FSMs (e.g. the statecharts family [Har87] [Har90]), or at least through a HDL (basically VHDL and Verilog).
- Some parts can be designed without any particular bias, currently in our system this is realized by using a process algebra computational model; however different solutions can be envisaged without loss of generality.
- Non purely functional requirements, such as area, total cost, power are given by considering the system as a whole, regardless of its modules composition and unbalanced granularity. This sometimes catches the co-design process in local optimums, because design optimization is usually performed onto a limited subset of the modules composing the system.

To adjust to such an industrial practice, the methodology developed in TOSCA allows the concurrent existence of multiple formalisms able to cover hardware oriented models, intrinsically software parts as well as specification parts without any particular implementation bias. To manage such a variety of formalisms, a uniform internal representation has been created based on a process algebra computational model with an OCCAMII more pragmatic syntax [Jpb94] [Bal96]. Modules which are already synthesized and cannot be modified in the following phases of system exploration are only encapsulated, while other modules described through different formalisms can be mapped (up to now only speedCHART and OCCAMII, but any formalism can be easily translated into the formal internal representation adopted). The overall system representation is stored within an object oriented database (TOSCA DB) tailored to support high-level architectural exploration, shared by all the tools composing the TOSCA environment.

System designs can also be specified by means of a graphical front-end for OCCAMII specifications developed in TCL (Tools Command Language). Three different editors have been developed to manage the design complexity.

1. A Project editor/Exploration Manager, that is the user interface aiming at providing a complete management of the co-design flow (e.g. saving/restoring of different versions of the project, documentation of the analysis results, ...); fig.2 depicts the graphical front-end gathering all the tools composing the TOSCA environment.
2. A Process editor, allowing the specification of graphical state-based processes, including an editor for the textual parts; fig.3 shows the capturing of a system description through the built-in process editor.
3. A Hierarchy editor, whose main purpose is to manage the specification of a complex system in terms of connections between hierarchical modules, possibly belonging to heterogeneous domain (e.g. existing VHDL library models, graphical FSMs descriptions,...)

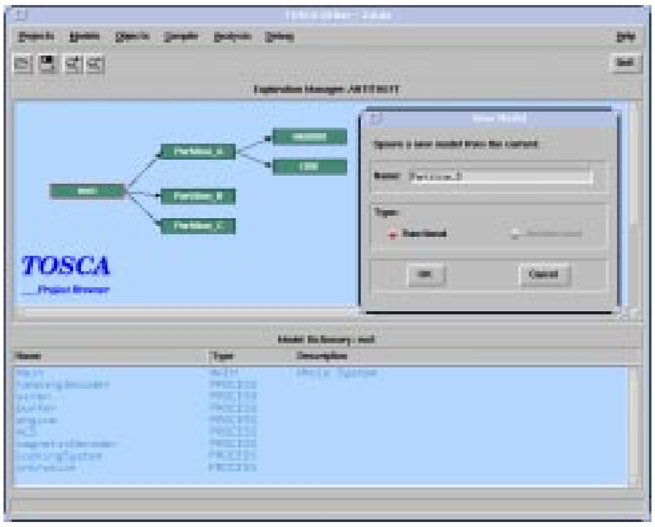


Figure 2.The Project Editor framework.

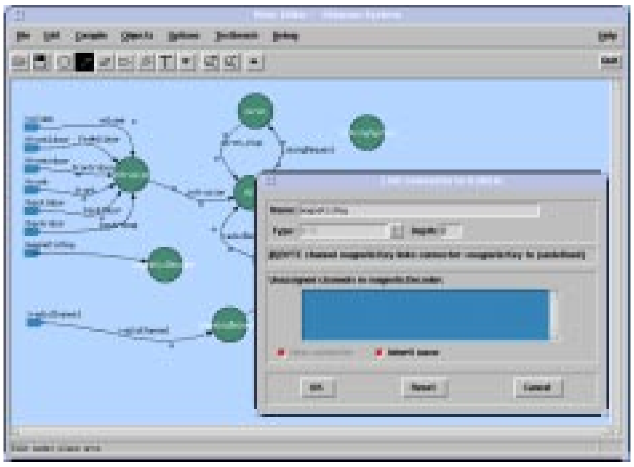


Figure 3. The Process Editor.

A simulation tool based on Petri Nets, written in C, has been developed for early functional validation and profiling.

The entire co-design process evolution is controlled by an Exploration Manager (EM) tool whose goal is to maintain a complete history of the multiple alternative design paths explored by the user. All the data are stored within a common design database whose user-friendly interface enables direct intervention in each of the steps composing system-level design, simulation and synthesis.

The EM allows the user to manage:

- The internal representation of the system specification;
- The evaluation of different design alternatives at system level;
- A step by step visualization of statistics and results concerning the transformations performed onto the system representation;
- The documentation of the design flow spanning from the system specification to the synthesis data.

After the preliminary phase of design capture, the main activity managed by the EM is the manipulation of the initial system modularization to produce a new set of system partitions and their association (if still *floating*) either with software or dedicated hardware units. Our approach, apart from possible bindings forced by users, does not start from an *a priori* default solution (e.g. initially fully software bound).

The system exploration and partitioning process can be viewed as an incremental activity of modification of the initial specification through the application of transformations. The TOSCA project has considered this process through:

- The definition and implementation of formal transformation rules working onto the internal model stored within the design database [For95] [Bal95];
- The definition of a partitioning algorithm and its implementation within the Exploration Manager framework to obtain a tool allowing both direct intervention of the user and automatic selection of the strategies by following the built-in evaluation criteria [Bal95];
- The definition and implementation of a set of metrics to drive the partitioning process [Bal95] [For95] [Bal96].

The user can organize these actions either along customizable schedules called *recipes* or can proceed under his own direct control. The output of this activity is a set of monolithic architectural units with a binding establishing either a hardware or a software implementation. Each architectural unit is then passed as input to the following co-synthesis stages.

The criteria guiding the selection of the transformations to be applied to the system description (in order to fulfill the target design requirements) are supported by a quantitative evaluation. Such an evaluation is performed by means of a set of metrics which drive both the definition of a coarse-grain initial solution and an iterative fine tuning until all the design constraints are met.

The basic transformation applied is processes collapsing. This will occur whenever the measure of a closeness criteria, during the selection of the closest processes, will remain under a certain *a priori* defined threshold. This solution is particularly flexible since it is possible to consider a set of criteria, with different priorities that will be changed dynamically, according to the current processes granularity or user choice.

The activity of identifying some closeness properties among parts of the system specification, can be performed as a static analysis based on metrics for early prediction of cost/performance. It aims at producing an initial nearly-optimum allocation and binding to be iteratively improved by the user until design goals are satisfied. The final exploration stage, including at least one actual synthesis cycle, is the most time consuming and it is strongly sensitive to the quality of the pre-allocation performed during the former phases; it returns information that will be back-annotated as a replacement of predicted data.

The metrics consider the system analysis from a threefold point of view:

- *Statically,* by analyzing composition and structure of the description contained within the database.
- *Dynamically*, but still independent of the implementation, through an high-level execution and profiling of the specification in order to extract information such as communication bottlenecks and statistics on operators applications to better tune the decision on the initial partitioning and binding.
- *After-synthesis*, requiring at least a complete synthesis cycle. This returns information that will be back-annotated as a replacement of predicted data and will also contribute to the final design evaluation.

A proper set of metrics for early prediction of cost/performance, as well as to evaluate the system synthesis results, has been developed and constitutes the basis for the partitioning process. The following types of parameters have been considered in the definition of the cost/performance metrics.

- *Communication*: costs related to the number of lines and bandwidth for hardware-hardware and software-hardware communication.
- *Interfacing*: according to the adopted communication/synchronization technique among different modules (e.g. between hardware and software, one interface unit per coprocessor vs. a common bus manager/arbiter queuing messages), costs can be affected by number and granularity of modules. Alternative bus protocol templates can be evaluated.
- *Area*: this is the most important aspect because of the single chip implementation. Overall optimization takes into account possible user-defined binding along the evaluation and comparison among different alternatives.
- *Resources exploitation*: on the software side, since the microprocessor is anyhow present, it is important to increase as much as possible the CPU utilization while fulfilling the temporal requirements of the programmed modules and the effectiveness of memory. Another relevant issue is related to the power consumption that is improved by a modularization able to identify the minimum set of coprocessors with the lowest amount of idle time.
- *Reuse*: expressed by the number and complexity of different subsections used to build the modules and by the quota of the system covered by library components or existing modules.

More specific information on the definition and use of metrics can be found in [Bal95] [Bal96], since it does not represent the main focus of this paper.

The co-synthesis step is constituted by two different tools: a tool for VHDL description generation for hardware-bound architectural units including interface generation, and a software synthesis tool for software-bound architectural units, including the Operating System support.
The class of embedded systems we are considering, characterized by real-time reactive requirements, and the typical application size enabling the use of a single ASIC including a CPU, has led to discard a C-language based solution for the software parts since we believe that software needs to be considered at a lower level, to carefully control time delay, code size and low level interfacing alternative schemes. Easy retargeting and portability are frequently advocated as some of

the main advantages of high-level languages with respect to assembly, our solution is to consider the software description at the level of a virtual assembly instruction set (VIS) whose structure can be mapped onto different CPU cores with fully predictable translation rules and, consequently, reliable performance estimation. As shown in the following, this solution provides the possibility of achieving a fully VHDL based co-simulation of each proposed hardware/software architecture, in order to get feedback on the effectiveness of the implementation. In fact a suitable VHDL model for VIS instruction-level execution, has been developed [Ant94b], so that the entire system can be co-simulated through a unified VHDL-based environment. The model is parametrizable in order to make possible the analysis of low-level timing/cost/performance, for different classes of microprocessors.

Up to now studies devoted to include a C-based real-time preemptive microkernel have been postponed. According to the needs of the application field, requiring simplicity and predictability, a static schedule approach with a coroutine scheme has been adopted as the target software structure.

Hardware synthesis is performed by generating behavioral VHDL for synthesis from the internal OCCAMII representation of the selected modules. It is also possible to generate RTL level VHDL for logic synthesis for finite state machine descriptions. Interfaces are generated with respect to the adopted model of communication. At this time a fixed protocol is implemented, coherent with the target architecture adopted as discussed in section 3.

3. System synthesis

In our approach, the system is intended to be implemented onto a single chip including an off-the-shelf microprocessor core with its memory (even if part of the memory can be external) and the dedicated logic implementing a set of coprocessors, i.e. the set of synthesized hardware-bound modules identified during system exploration (see fig.4).

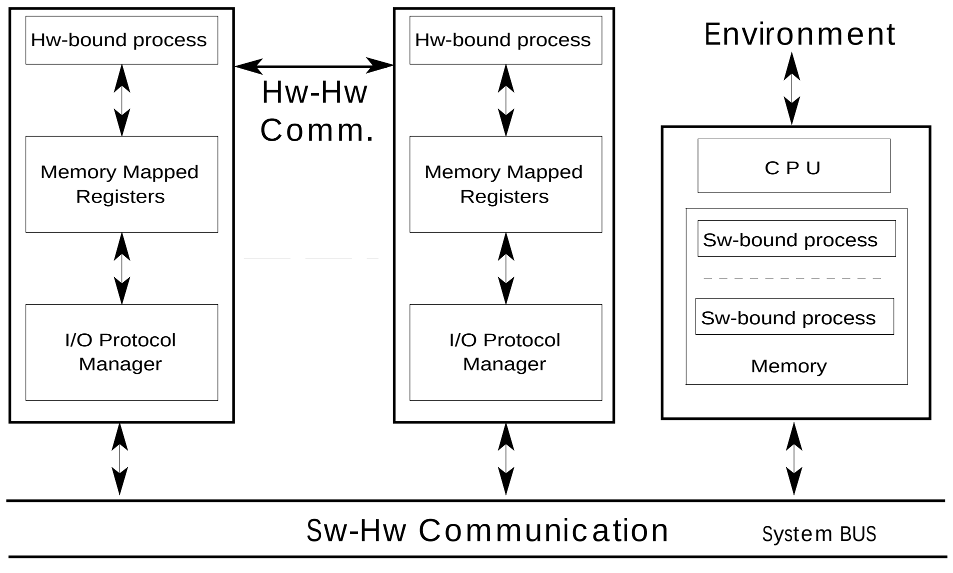


Figure 4. The hardware-software target architecture.

In this discussion the term coprocessor includes also arithmetic/logic operations and possible private storage capability, while high-level synthesis tools typically separate controllers from data-paths. The master processor is programmable and the software can be either on-chip resident or read from an external memory; dedicated units operate as peripheral coprocessors. Hardware and software bound elements are interfaced by means of a master-slave shared bus communication strategy. All hardware to hardware communications are managed through dedicated lines. The RAM memory required for program/data storage shares the main data bus with the coprocessors, but can be accessed only by the master CPU. Communications among CPU and coprocessors are based on a memory mapped I/O scheme with one bus interface manager per coprocessor based on a common I/O buffered protocol manager.

Before committing to a specific implementation, the initial system specification can be manipulated to fulfill the target design requirements. This partitioning process, can be viewed as an incremental activity of modification of the initial specification through the application of transformations [For95] [Bal96], briefly summarized in section 2.

Once a pair of hardware and software bound sets of modules have been defined, the following step is to produce their implementations. The synthesis stage will produce a mapping of the system onto the target architecture reported in fig.4, i.e.:

- assembly-level code for each sw-bound process, according to the target microprocessor instruction set;
- operating system support for process to process communication (both between sw to sw and sw to hw), as well as for CPU scheduling;
- VHDL code for each architectural unit (coprocessor) corresponding to hw-bound processes; this includes also the implementation of the hardware side of the interfacing subsystem, allowing the mapping of the abstract process to process communication onto an actual system architecture.

3.1 Software synthesis

The software system has to be designed according to the reference architecture that is itself strongly influenced in terms of programming paradigm and hardware by the application field and the cost/performance goals. The basic requirements of an embedded system are the performing of activities according to a set of precise timing constraints (*timeliness*) and the *flexibility*, that in our case means that system configuration and software behavior have to be easy to update. These requirements are crucial to enhance maintenance, possibility of customization and re-use of the system and design methodology. Low-cost embedded systems characterized by small/medium size applications require the development of a light-weight software in terms of typical operating system services provided, but with an high degree of reliability and predictability. The run-time support provided in TOSCA has been kept minimal and includes only those features that are actually needed to support exception handling, configuration control, communication management and process activation, chosen during the customization phase. The operating system micro-kernel actually acts as a high-level process manager whose evolution is controlled by a deterministic algorithm, with synchronization among processes or with the environment (i.e. the coprocessors or external devices connected to the system).

Since the current target architecture considers just one microprocessor, concurrence is emulated through interleaving of processes, each corresponding to a software-bound part of the system modularization, whose ordering is statically defined, i.e. a pre-runtime schedule has been adopted. This solution has been chosen because high processor utilization is foreseen to reduce implementation costs, so that there is not much spare CPU available. As a consequence, a solution able to guarantee *a priori* that all the stringent timing constraints will be met, seems to be the only viable. We found many advantages of this solution compared to the presence of an on-line schedule policy, such as the significant reduction in the share of run-time resources necessary to implement context switching and the scheduling itself, but the most important is that it is easier to satisfy our primary goal of meeting the real-time deadline.

Software-bound processes, that are viewed as a set of sequential cooperating threads with shared memory similar to a coroutine scheme, are constituted by operations that must be executed in a prescribed order. The number of processes is known in advance, and it will never change run-time. This implies that the operating system does not require a dynamic scheduling since the scheduling policy can be computed off-line and *code-wired*. Therefore, the solution proposed requires only a small operating system providing the mechanisms for process activation and the communication support.

In general, two classes of processes can be present: *periodic*, whose computation is executed repeatedly in a fixed amount of time and *asynchronous*, that usually consist of computations responding to an event (internal or external). A typical example of periodic process is the sampling of external data with the consequent updating of state internal variables and outputs as it happens in a transmission frame manager of a telecom digital switching system. For a number of real-time applications, periodic processes where the sequencing and timing constraints are known in advance, seem to constitute the bulk of computation. The number of asynchronous processes is usually limited, requiring short computation times. Moreover, information due to external or internal events can be recorded and buffered until they can be handled by periodic processes tailored to serve them. Different and more general techniques for mapping asynchronous processes onto an equivalent set of periodic processes can be found in [Mok84], therefore we considered only the problem of scheduling periodic processes.

The algorithm we implemented, given the set of processes constituting the sw-bound part of the system, determines a schedule (whenever it exists) such that each process is activated after its release time and carries out its computation before its deadline. Fig.5 reports the timing characterization of a periodic process. Even though the exact timing characteristics of system components and events sometimes cannot be predicted, we overcome such a problem by using a worst case estimation of these parameters so that the scheduling algorithm can guarantee a predictable behavior.

The methodology we adopted to obtain the pre-run time schedule is based upon a systematic improvement of an initial schedule until a feasible (near optimal) schedule is found. The analysis of the VIS code corresponding to each software-bound part allows the VIS scheduler to consider each software process characterized by a *release* time, the *duration* and a *deadline*, which can be broken into a set of code segments. An analysis of the internal composition of the process provides the start time of each code segment relative to the beginning of the process it belongs. Exclusion relations

may be present among segments when some of them must avoid interruption by others to prevent possible errors caused by simultaneous access to shared resources, e.g. data structures, I/O devices, coprocessors. Precedence relations, that occur when a segment requires some information produced by other process segments, are also considered.
The scheduler produces an ordered set of code segments fulfilling deadlines and constraints (if they exist) where the lateness of all segments is minimized. The context switching overhead has been considered by including an additional delay to the purely computational time. To characterize the software running onto the microprocessor, other information is produced by the scheduler by inspecting the final schedule produced: the level of process fragmentation and the relative overhead, the slack time, the CPU utilization and, in case of non feasible schedule, the critical segments responsible of the algorithm failure.

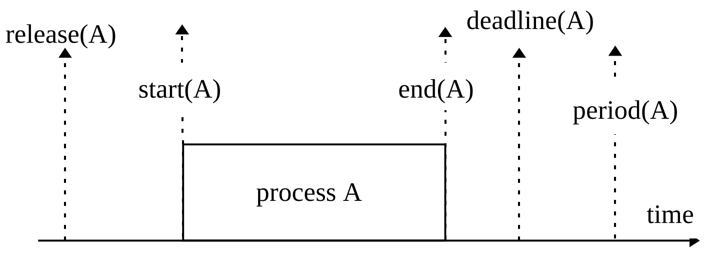


Figure 5. Timing characterization of a process.

The starting point of the algorithm is a valid initial schedule satisfying release times, exclusion and precedence constraints obtained through an heuristic belonging to the class *earliest-deadline-first*. Through a branch and bound technique a search-tree is built where each node is a schedule. Nodes are obtained from their parents by introducing new additional properties (preemption and exclusion): by computing the lower bound of the lateness of any schedule deriving from that node the best solution among all candidates is selected. A skeleton of the algorithm is depicted in fig.6.

```
repeat {
    select child-node with min-lateness among unexpanded nodes;
    compute SG1, SG2;
    for each segment ∈ SG1 and SG2 create new schedule;
    compute lateness lower-bound of each schedule;
until (lateness - LowerBound) ≤ ε }
```

Figure 6. The scheduling strategy, ε is the maximum acceptable error.

A *valid solution*$_i$ (VS$_i$) computed through an earliest-deadline-first policy and its latest segment (LS$_i$) are associated with the generic node *i* of the search tree. In order to produce new solutions improving the current one, two groups of segments SG1$_i$, SG2$_i$ are determined such that:

a) SG1$_i$: VS$_i$ can be improved if the latest segment is *scheduled before* a segment of SG1$_i$;
b) SG2$_i$: VS$_i$ can be improved if the latest segment *preempts* a segment of SG2$_i$.

According to the properties a) and b), for each segment belonging to the two groups SG1 and SG2, a new schedule (successor node) is created. The result of this step consists of two sets of new schedules (nodes), that are dominated by the node$_i$, associated with the lower bound on the lateness for each schedule obtained during this stage. Among the unexpanded nodes, the one with the lowest bound is considered the best candidate to achieve an optimal solution (i.e. lateness equal to a goal value `LowerBound`) for a new branching. The search proceeds until either a feasible solution is determined or no unexpanded node exists with a lower bound less than the least lateness of all the valid solutions computed up to now (that will constitute an optimal solution). In such a way, all the possible improvements of the initial schedule are considered, so that the optimal solution is determined, if one exists. Moreover, in the case of algorithm failure, the responsible segment (the one with the least lateness) and consequently the owner process is determined and can be used as starting point for actions aiming at making the implementation of the software-bound parts feasible.

As an example let us consider the simple case of the following input file (left) for the scheduler and the corresponding graphical representation (right):

```
{Pa 0 30 80 0 0} {} {}
{Pb 20 20 81 0 0} {C} {}
{Pc 40 30 70 0 0} {} {}
```

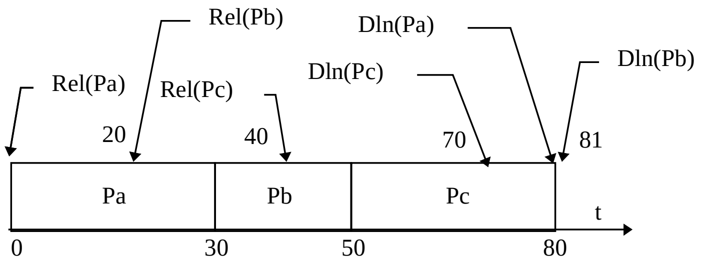


Figure 7. An example of scheduling problem associated with its textual and graphical representation. Rel(P) and Dln(P) indicates *release* and *deadline* of the process P.

The problem is composed of three processes (Pa, Pb, Pc) where, the items of the corresponding list represent the name, the release time, the computation time, the deadline and two values to mimic the delay necessary to restore and save the process context (for simplicity set to zero), respectively. The other lists report possible constraints (exclusion and preemption), in the proposed example `Pb excludes Pc`. The expanding sets are SG1={Pb} and SG2={Pa}, therefore the two sets of new schedules are composed only by one element each. The actual schedules are obtained from the characteristic properties of SG1, i.e. {Pc precedes Pb} (fig.8-left) and similarly for SG2, i.e. {Pc Preempts Pa, Pb Preempts Pa} (fig.8-right).

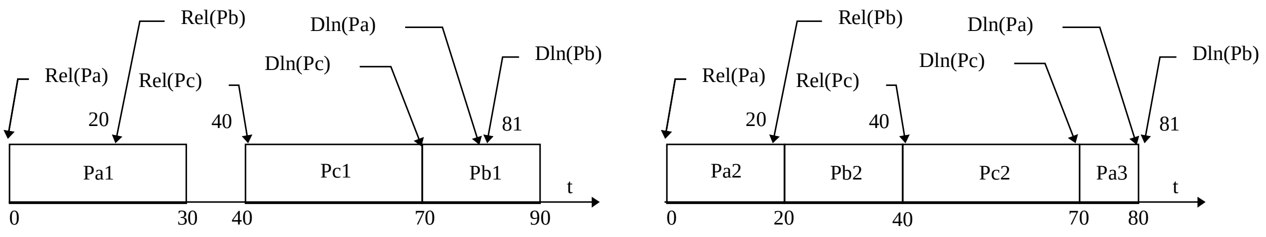


Figure 8. The schedules originated by the SG1 and SG2 expanding sets.

For each schedule the report file includes a section containing the information summarized in fig.9.

In the leftmost schedule the latest segment is Pb1 and the Lateness evaluates End(Pb1)-Deadline(Pb1) = 9 that is a local optimum. The latest segments of the second open node of the search tree, i.e. the rightmost schedule, are Pa3 and Pc2, the Lateness evaluates End(Pa3)-Deadline(Pa) = End(Pc2)-Deadline(Pc) = 0. No further expansion of the tree is possible, therefore the rightmost schedule represents a global optimum since Lateness is equal to least `LowerBound` of all the open nodes considered up to now. The output of the scheduler is thus the process segmentation and ordering on the right of fig.8.

	Schedule of fig.8 (left)	Schedule of fig.8 (right)
`ListSortedSegments`	`Pa1, Pc1, Pb1`	`Pa2, Pb2, Pc2, Pa3`
`Sections`	`Pa1, Pc1, Pb1`	`Pa2, Pb2, Pc2, Pa3`
`LatenessSchedule`	`9`	`0`
`LatestSegment`	`Pb1`	`Pa3, Pc2`
`CPUinactivity`	`10`	`0`
`MaxSlackTime`	`10`	`0`
`EndSchedule`	`90`	`80`
`StartSchedule`	`0`	`0`
`TotalTimeSchedule`	`90`	`80`
`LowerBound`	`9`	`0`

Figure 9. Data computed at each step of the scheduling process. `LatestSegment` is the critical parameter determining the lateness of the schedule, `TotalTimeSchedule` is the overall execution time of the schedule and the `Sections` field contains the sets of contiguous segments.

The software part of the system is implemented by means of a generic Virtual Instruction Set (VIS) which allows a better control of time delays, code size and a low level characterization of I/O interfaces. The VIS is an intermediate language between OCCAMII and the target CPU assembly aiming at capturing the minimum set of features shared by microcontrollers for embedded applications. This solution allows us to achieve the following goals:

- integrated simulation of the mixed hardware-software system;
- extension of the analysis to cover also multiple processor families;
- good predictability of the final running software behavior;
- integrated synthesis flow able to cover the entire system development in terms of hardware, interfaces, software and operating system support.

The VIS is defined in terms of a customizable and orthogonal register-oriented machine with a common address space for both code and data. This means that each register can act as accumulator and all the operations (e.g. addressing, arithmetic-logic, data transfer) can be performed no matter which register is used as operand. The instruction set has been designed in order to be easily retargeted onto different CPUs: a mix of CISC and RISC typical instructions are included. A generic VIS instruction can either be one-to-one mapped on a native target assembly instruction or correspond to a group of assembly instructions. In such a way, if the selected CPU does not match the VIS instruction, the retargeting of the code is performed via an alternative definition of the instruction using only the RISC-*side* of the VIS, thus reducing the effort to reconfigure the software whenever alternative CPUs are evaluated.

The VIS supports unsigned/signed integer data types (BIT, BYTE, INT16 or *word* and 32-bits integer called *longword*) as well as all typical arithmetic/logic operations. The address space spans over 32 bits so that each VIS argument is always contained within a longword. The memory format

for the data is aligned in terms of 32-bits words, as a consequence four memory locations are necessary to store a byte. Boolean variables can be packed to save memory space.
The instruction format is similar to the one of the MC68000 with a suffix indicating the operand type, e.g. `MOVE.B R1, R2` copies a byte from `R1` to `R2`. Three types of instruction formats have been defined:

- `op destination, source`
- `op arg1, arg2`
- `op`

where both `destination` and `source` can be registers or memory references (*source* can also be an immediate operand) and `arg1,arg2` model operations where the argument order is not relevant.
The generation of the VIS code as well as the final implementation of the software running on the target microprocessor follows the phases depicted in fig.10. Three main steps compose the top part of such an activity: initialization, code generation, estimation of time delays and binary code size.

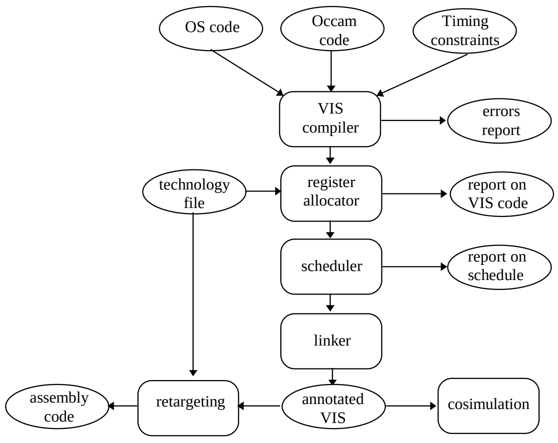


Figure 10. The software generation process.

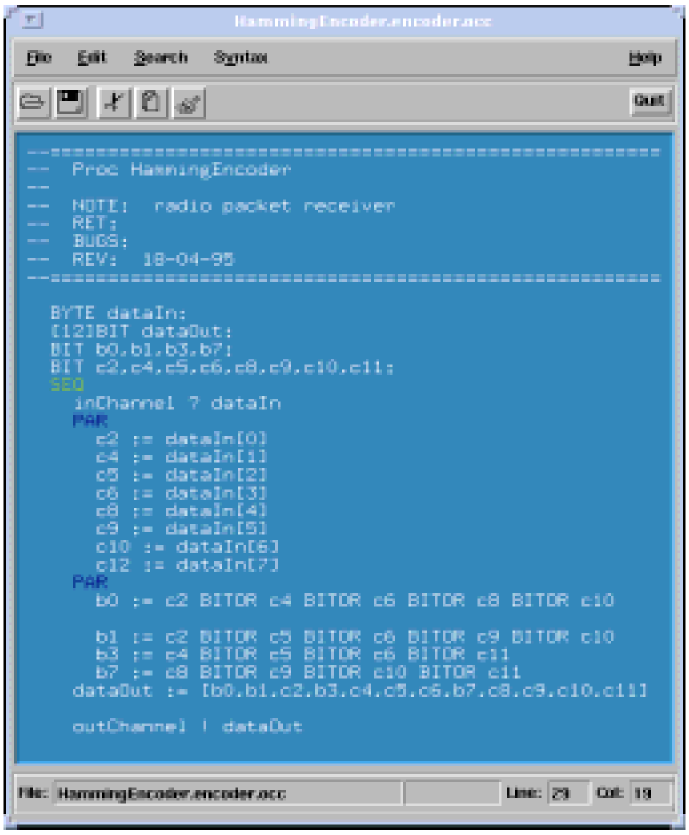
At the beginning the system is initialized by reading from a technology file the information characterizing the selected CPU, e.g. the clock period, the instruction set, the registers number, type and size of the registers, the number of clock per instruction, etc. . The code generation involves register usage optimization and automated packing of bit variables. The estimation of timing performance and memory requirements for back-annotation towards the system design exploration phase can be obtained for several possible microprocessor cores. In fact, its actual behavior is represented by the following three groups of information (for each foreseen CPU):

- retargeting rules (RR): specifying the rules for mapping VIS code onto the target microprocessor instruction set;
- time/size table (TST): reporting for each VIS instruction the number of clock cycles and bytes of the corresponding target CPU mapping (which, in general, is not composed of a single instruction).
- technology (TF): containing information on the adopted CPU as the BUS width, the power consumption, the pin-out of the microprocessor, the particular characteristics of the adopted

model of microprocessor with respect to the rest of the CPU family, such as, for instance the memory size.

The first step is the compilation of the OCCAMII specification in VIS. Although the entire system specification will not probably be implemented in software, the estimation of the VIS performance and cost is initially carried out for all the OCCAMII modules composing the description. The obtained result is employed during the system partitioning to compare/drive alternative modularizations and hardware vs software bindings. The obtained code is not executable since the following decisions have yet to be taken: register mapping, process scheduling, system bootstrap, memory allocation including symbolic vs actual address determination.

A pre-allocation of the register to extract execution times and memory requirements is performed according to the information included within the technology file. The VIS code is then annotated with the information needed by the scheduler to produce a correct ordering of the processes execution, by adding some bracket-encapsulated tags. A simple example of VIS compilation for a sub-part (Hamming encoder) belonging to a car antitheft system that we use as a small benchmarking, is shown in appendix 1. The example is composed of a process receiving data from the channel `dataIn` where two parallel sections allow the system to compute the Hamming encoding of the input to be transmitted on the `dataOut` channel (see fig.11).



```
--==========================================================
--   Proc HammingEncoder
--
--   NOTE:   radio packet receiver
--   RET:
--   BUGS:
--   REV:   18-04-95
--==========================================================

   BYTE dataIn:
   [12]BIT dataOut:
   BIT b0.b1.b3.b7:
   BIT c2.c4.c5.c6.c8.c9.c10.c11:
   SEQ
     inChannel ? dataIn
     PAR
       c2 := dataIn[0]
       c4 := dataIn[1]
       c5 := dataIn[2]
       c6 := dataIn[3]
       c8 := dataIn[4]
       c9 := dataIn[5]
       c10 := dataIn[6]
       c12 := dataIn[7]
     PAR
       b0 := c2 BITOR c4 BITOR c6 BITOR c8 BITOR c10

       b1 := c2 BITOR c5 BITOR c6 BITOR c9 BITOR c10
       b3 := c4 BITOR c5 BITOR c6 BITOR c11
       b7 := c8 BITOR c9 BITOR c10 BITOR c11
     dataOut := [b0.b1.c2.b3.c4.c5.c6.b7.c8.c9.c10.c11]

     outChannel ! dataOut
```

Figure 11. Screen-dump of the description of a Hamming decoder described by using the TOSCA OCCAMII process editor.

The VIS code maintains a structure similar to the original OCCAMII model: the body of each process is identified through the `<process>` and `</process>` tags while concurrent processes (corresponding to the PAR construct of OCCAM) fall within the scope of a `<group>` identifier.

As reported above, the scheduler may require to break the processes to meet time deadlines; as a consequence, it is necessary to consider the impact of additional context-switching overheads. The scheduler performs such an analysis by considering the `<USE Regs-List>` and `<LEAVE Regs-List>` tags which represent, incrementally, the registers necessary to be saved at any point in time. Critical sections, corresponding to non-breakable actions such as an interrupt handling, are enclosed within `<atomic> </atomic>` to prevent a possible preemption.
Data transfer from software to hardware and viceversa is modeled via memory-mapped coprocessor registers, associated with each port. In this example, the 12-bits channel has been mapped onto a 16 bits word corresponding to a pair of contiguous memory locations.
Timing characterization is also performed with a fine granularity to improve the freedom of the scheduler to choose the point where to break processes. The left-most tags of each VIS instruction contains the computation of `<minimum typical maximum>` delays to execute the operation according to the target CPU. Up to now, according to the most common types of embedded system microprocessors, effects concerning pipelined instruction execution or parallel fetch have not been considered. For an analysis on how these issues can be managed for DSP applications, see [Sut93].
During software synthesis, processes as well as the operating system microkernel are directly assembled into VIS code. As reported above, the software system is composed of processes and of a kernel basically operating as a context switcher: although no sophisticated mechanisms for memory protection are necessary, particular attention has been devoted to the software section responsible for communication by adopting *ad-hoc* solutions to suit each specific circumstance. Our software synthesis system has to implement two different communication schemes: software to software, hardware to software (and viceversa). Processes communication takes place through buffered channels that will be implemented according to the type of data protocol and the hw/sw binding of the source and target processes.
Protocol implementation of complex data types is defined in terms of composition of basic types, such as BOOL, BYTE, INT16 (16-bit integer). The needs for communication involving the system bus have been reduced during the system partitioning phase since, under the scheduling algorithm viewpoint, the bus is a shared resource that will originate a *critical section* within the software-bound process requiring its use, thus increasing the difficulty to determine a feasible schedule.
Even though the basic OCCAMII model is composed of direct, point-to-point, asynchronous channels, our implementation has been extended to provide also a broadcasting node by expanding its definition into a software process able to copy the datum on all the target channels. The channel is mapped onto a pair {memory variable, data ready flag} shared by all processes. Since communication rates can vary across different processes, no matter if they belong to the same hardware or software partition, appropriate FIFO buffering capability has been introduced. Hardware/software interface is performed via memory mapped registers.
A parametrizable retargeting tool, able to map VIS code on different target CPU has been implemented and tested for a Motorola 68000 microprocessor family, the extension towards the PowerPC architecture is part of the current effort.

3.2 Hardware and interface synthesis

Concerning the *hardware mapping* strategy adopted in TOSCA, it should be pointed out that control-oriented specifications cannot easily be managed by classical high-level synthesis approaches involving operators scheduling. In fact, circuit speed estimation is very difficult when dealing with descriptions dominated by conditional functions, where arithmetic operations are typically restricted to a few sums and comparisons (if anyone of them is present at all). During the next stage involving VHDL translation into a generic netlist, technology mapping and logic implementation, any direct relation 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 previous version of the TOSCA module devoted to hardware mapping, each hardware-bound architectural unit is implemented by generating a finite state machine VHDL description, together with its bus interface [Ant94b]. If the starting point is a synchronous model (as those obtained from speedCHART), no additional scheduling step is needed. The VHDL code generator translates the internal representation of each FSM into a VHDL template (block-encapsulated processes) compliant to the guidelines for synthesis enforced by commercial tools such as Mentor Graphics Autologic and Synopsys VHDL Compiler. 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 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.

However, recently several commercial VHDL behavioral synthesis tools are emerging (Synopsys, Synthesia). This opportunity is particularly valuable to cope with system-level architectural exploration needing fast speed/cost prediction techniques, avoiding as much as possible to move down to the gate-level netlist. The current version of TOSCA is oriented towards a direct compilation of process algebra description into behavioral VHDL, that is becoming the target abstraction level for synthesis. In summary, three different hardware synthesis paths can be applied:

1. transparent passing of the initial VHDL description to the synthesis tool;
2. RTL synthesizable VHDL description of the modules such as FSM;
3. behavioral VHDL descriptions of the hardware-bound modules.

A suitable VHDL generator has been developed, starting from the OCCAMII description stored within the database and building a tree modeling the statements nesting. It produces a set of modules corresponding to the hardware bound architectural units (coprocessors) with their communication interfaces. The VHDL code generator performs a depth-first scan of the tree representing the OCCAMII structure and produces two output files: the first contains the entities declarations with the corresponding behavioral description while the second is a package containing all the procedures necessary to realize the communication among processes.

Since channels are not supported by VHDL, *ad hoc* fully hardware interface structures covering both buffered and unbuffered communication have been introduced.

In case of hardware-to-hardware communication, the realization of each channel requires the instance of three signals to implement the negotiation process - `s_req` (send-request), `s_data` (send-data) and `r_ack` (receive-ack) - that will be used to expand each OCCAMII declaration of channel - `CHAN OF TYPE ChannelName` - into the following VHDL code:

```
signal ChannelName_s_req:boolean:=false;
signal ChannelName_s_data:=0;
signal ChannelName_r_ack:boolean:=false;
```

These variables are used to generate an handshake mechanism with the semantic defined by the following VHDL code:

```
if not s_req then
      wait until s_req;
else
      wait for 0 ns;
end if;
```

The OCCAMII statements for message passing *ChannelName!variable* and *ChannelName?variable* are translated into the following two procedures, respectively:

```
send_unbuffered(ChannelName_s_req,ChannelName_r_ack,ChannelName_s_data,variable)
recv_unbuffered(ChannelName_s_req,ChannelName_r_ack,ChannelName_s_data,variable)
```

The unbuffered communication needs an additional pair of signals, `full_f` (fifo full) and `empty_f` (fifo empty), modeling the availability of free positions within the FIFO buffer, as input to the sender and the receiver processes, respectively. The VHDL procedure for buffered communication thus becomes:

```
send_f (ChName_s_req, ChlName_r_ack, ChName_s_data, ChName_full_f, variable)
recv_f (ChName_s_req, ChName_r_ack, ChName_s_data, ChName_empty_f, variable)
```

In addition to the above procedure, a VHDL process *fifo_ChName* will be instanced to actually implement the message storage and management unit. This element can be customized according to the desired buffer size (N) and channel type (e.g. INT).

Each entity description of the modules connected via hardware channels contains the declaration of a set of ports corresponding to the signals used to implement the communication protocol.

The hardware side of the hw-to-sw communication has been implemented by studying an additional BUS interface unit to be added to the coprocessor (see fig.4). Such a module contains a pair (input and output) of FIFO buffers used to store the messages that processor and coprocessor need to exchange. The entire communication is mastered by the processor which triggers the reading of data from the output queue or the sending of messages to the input queue according to the data flow direction of the original OCCAMII channel. When a message is sent out on the BUS, the coprocessor protocol manager performs a decoding of the BUS address to discover if it has to be processed. A maximum of 255 coprocessors are allowed with at most eight bi-directional channels per each. For each queue, a pair of status registers (in, out) storing the information on FIFO content (e.g. empty) are foreseen, their information can be accessed by the processor communication primitives. In summary, for the target architecture we are considering, the ADDRESS BUS bits have been associated with the following information:

A0..A2	coprocessors or other peripheral selection;
A3	data BUS contains a datum or the status register;
A4	selection of the status register to put on the data bus;
A5..A7	FIFO selection among the possible eight per coprocessor;
A8..A15	coprocessor selection among the possible 255.

The customization of such a scheme onto the MC68000 is straightforward, the only additional control signals to be taken into account are R/W, BUSREQ and MEMREQ. When a software-bound process needs to send a datum to the *n-th* coprocessor, the software communication procedures will put on A8..A15 the binary encoding of the coprocessor number, on A5..A7 the FIFO identifier corresponding to the channel considered, A0..A2=1 to select the coprocessors address space, A3=0, R/W=0 and MEMREQ=BUSREQ=1. The datum on the data bus will be acquired by the addressed coprocessor and placed within the proper input FIFO queue which is assumed to be sufficiently large to contain all the incoming messages (this is a matter of correct design, not impacting the suitability of the proposed scheme). The hw-to-sw communication takes place in a similar way.

4. Simulation of interacting hardware and software subparts

Although a preliminary validation of the initial system-level specification is performed during the initial design phases, an additional simulation step at the hw/sw architectural level still represents a significant value-added to obtain feedback 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 co-simulation and logic synthesis stages only.
The co-simulation 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 for the dedicated coprocessors and interfaces;
- VHDL language features allow 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 co-design methodology.

Co-simulation 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 co-simulation. 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.
For simulation purposes only, each target CPU core model will be implemented through

- a basic *kernel* executing VIS code that is target-CPU independent;
- a customizable target-dependent I/O module, tailored to manage the bus-based interface to/from the dedicated coprocessors;
- the time/size table (TST) of the selected target CPU.

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.
For instance, the internal structure of the M68000 core model is shown in fig.12. The `vkernel` 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.

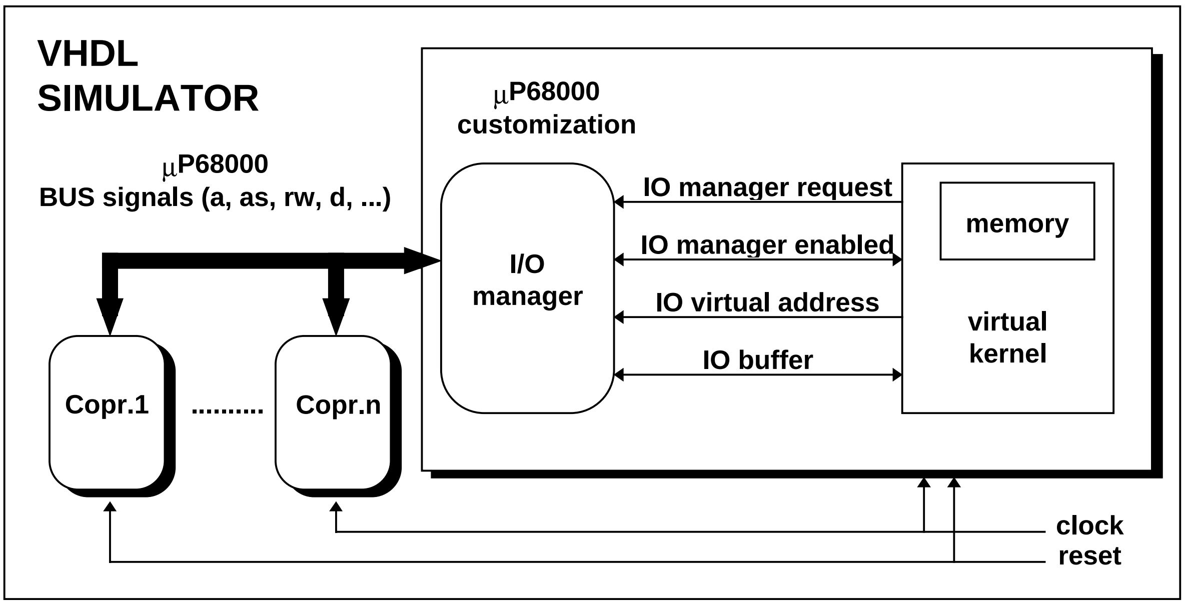


Figure 12. Virtual and target-specific parts in the CPU model and their connection with the rest of the system architecture.

A VHDL view of the entire hw/sw system for simulation underlying the architecture depicted on fig.12 is reported in fig.13; the Motorola 68000 has been adopted as target CPU core.

```
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 13. VHDL top-level representation of a mixed hw/sw system.

Each coprocessor 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.

An overview of the VHDL code implementing the entire CPU entity is presented in fig.14. The source code follows the modularization depicted in fig.12. The `io_manager` process communicates with the `vkernel` instance through the following signals:

`io_manager_enabled`:	used by the vkernel to enable the io_manager during the I/O stages;
`io_manager_request`:	specifies the kind of vkernel request (read or write);
`io_buffer`:	used for data transfer;
`io_virtual_address`:	used by the vkernel to communicate the VIS code addresses that must be translated 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<="zzzzzzzzzzzzzzzzzzzzzzz";
        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<="zzzzzzzzzzzzzzzzzzzzzzz";
      else -- write
        a<="zzzzzzzzzzzzzzzzzzzzzzz";
        rw<= ’1’;
        wait until clk=’1’ and clk’last_value =
’0’ and clk’event;
        a <=io_base +
std_ulogic_vector(io_virtual_address);
        wait until clk=’0’ and clk’last_value =
’1’ and clk’event;
        as <= ’0’;
        rw <= ’0’;
        wait until clk=’1’ and clk’last_value =
’0’ and clk’event;
        d <= io_buffer;
        wait until clk=’0’ and clk’last_value =
’1’ and clk’event;
        uds <= ’0’;
        lds <= ’0’;
        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;
        as <= ’1’;
        uds <= ’1’;
        lds <= ’1’;
        a<="zzzzzzzzzzzzzzzzzzzzzzz";
        rw <= ’1’;
        d<="zzzzzzzzzzzzzzzz";
        wait until clk=’0’ and clk’last_value =
’1’ and clk’event;
      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 14. VHDL code for the CPU entity.

The software mapping stage produces an ASCII file containing a virtual assembly code description. Such a file is loaded during the initialization phase of a simulation session into the internal data structures. Code and data segments are managed in different ways (fig.15).

```
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 15. 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 types for source operands are register (`reg`), memory (`mem`), bit vector/integer immediate (`bv_imm`, `int_imm`) or memory-mapped I/O (`io`), while destination types 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 parametrized through the DBANK_SIZE generic.
Instruction fetch/decode tasks are reported in fig.16. 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 vkernel_arc of vkernel 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 16. Instruction fetch/decode VHDL template.

To give a flavor of how the typical VIS instruction can be simulated, the corresponding VHDL source code is shown in fig.17. Such a 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 necessary to execute the instruction;
- 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 a certain number of clock cycles (specified by the BUS_READ_DELAY parameter) is expired. The delay value related to bus write operations is not needed since 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.

```
architecture vkernel_arc of vkernel is
...
procedure delay (opcode: in opcodes) is
begin
  for k in t_table(opcode'POS - 1) loop
    wait until clk='1' and clk'last_value =
'0'
         and clk'event;
  end loop;
end;
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 w_and =>
        if (i.dest.kind = reg) then
           case i.src.kind is
             when reg =>
               dbr(i.dest.int_value) <=
                 dbr(i.src.int_value) and
dbr(i.dest.int_value);
               delay(w_and);
             when mem =>
               dbr(i.dest.int_value) <=
                data(i.src.int_value) and
dbr(i.dest.int_value);
              delay(w_and);
            when bv_imm =>
              dbr(i.dest.int_value) <=
                i.src.bv_value and
dbr(i.dest.int_value);
              delay(w_and);
            when int_imm =>
              dbr(i.dest.int_value) <=
                i.src.int_value and
dbr(i.dest.int_value);
              delay(w_and);
            when io =>
              -- start bus read cycle
              io_manager_enabled <= true;
              io_manager_request <= read;
              -- bus transfer delay
              for k in BUS_READ_DELAY-1 loop
                wait until clk='1' and
clk'last_value = '0'
                     and clk'event;
              end loop;
              dbr(i.dest.int_value) <=
                io_buffer and
dbr(i.dest.int_value);
          end case;
        elsif (i.dest.kind = mem) then
          case i.src.kind is
            when reg =>
              data(i.dest.int_value) <=
                dbr(i.src.int_value) and
data(i.dest.int_value);
              delay(w_and);
            when bv_imm =>
              data(i.dest.int_value) <=
                i.src.bv_value) and
data(i.dest.int_value);
              delay(w_and);
            when int_imm =>
              data(i.dest.int_value) <=
                 i.src.int_value) and
data(i.dest.int_value);
               delay(w_and);
           end case;
        else -- i.dest.kind = io
          -- start bus read cycle
          io_manager_enabled <= true;
          io_manager_request <= read;
          -- bus transfer delay
          for k in BUS_READ_DELAY-1 loop
            wait until clk='1' and
clk'last_value = '0'
                 and clk'event;
          end loop;
          case i.src.kind is
             when reg =>
               io_buffer <=
                 dbr(i.src.int_value) and
io_buffer;
             when bv_imm =>
               io_buffer <=
                 i.src.bv_value and io_buffer;
             when int_imm =>
               io_buffer <=
                 i.src.int_value and
io_buffer;
           end case;
           -- start bus write cycle
           io_manager_enabled <= true;
           io_manager_request <= write;
           -- bus transfer delay
          for k in t_table(i.opcode'POS - 1) -
BUS_READ_DELAY loop
           wait until clk='1' and
clk'last_value = '0'
                 and clk'event;
           end loop;
        end if;
    end case;
  end process;
end;
```

Figure 17. VHDL implementation of a VIS instruction.

6. Concluding remarks

This paper has presented a suitable methodology to support hw/sw co-design. A prototype toolset covering co-speification, hw/sw architectural exploration, co-synthesis and co-simulation activities has been developed. The system achieves a good integration with the existing commercial design environments through the VHDL description of the hardware part, the assembly level synthesis of the software modules (and of the operating system support) and the import of design models defined via different design environments (e.g. speedCHART). The focus of this paper have been mainly the co-synthesis and co-simulation steps.

Software synthesis is performed by translating the software-bound architecture units into a Virtual Assembly code which satisfies all real-time constraints by means of an optimal static schedule. The Virtual Assembly is then translated into the actual microprocessor assembly chosen for the system. This solution allows an higher degree of control of the software execution and a greater flexibility in evaluating alternative microprocessor cores. Moreover, the fine granularity evaluation of the software characteristics improves the reliability of its cost estimation during the partitioning process with respect to a C-level analysis of the software parts.

The hardware modules are generated as VHDL code together with the interfaces which are, at this time, fixed. Different paradigms of communication will be made available in future versions of the framework.

Finally, co-simulation is achieved by completely modeling the hardware and software parts in a common VHDL-based environment.

Evaluation of these strategies have been performed on a number of medium size examples, allowing the identification of optimal solution in a reduced time. We are currently developing a large telecom example to test all features of the proposed approach.

7. References

[Alt91] M.Altmae, P.Gibson, L.Taxen, K.Torkelsson, *Verification of Systems Containing Hardware and Software,* in Proc. of EURO-VHDL '91, Stockholm, September 1991.

[Ant94a] S.Antoniazzi, A.Balboni, W.Fornaciari, D.Sciuto, *HW/SW Co-design for Embedded Telecom Systems,* in proc. of ICCD'94 IEEE Int. Conf. on Computer Design, pgg.278-291, Cambridge, Massachusetts, US, Oct. 10-12, 1994.

[Ant94b] S.Antoniazzi, A.Balboni, W.Fornaciari, D.Sciuto, *The Role of VHDL within the TOSCA Co-design Framework*, in Proc. of Euro-VHDL'94, September 1994, Grenoble, France.

[Bal95] A.Balboni, W.Fornaciari, D.Sciuto, *TOSCA: a pragmatic approach to co-design automation of control dominated systems*, Hardware/Software Co-design, NATO ASI Series, Series E: Applied Sciences - vol.310, pp.265-294, Kluwer Academic Publisher, 1996.

[Bal96] A.Balboni, W.Fornaciari, D.Sciuto, *System-level Exploration for Control-Dominated Embedded Systems*, in Proc. of APCHDL'96, Bangalore, India, January 1996.

[Ben93] T.Benner, R.Ernst, J.Henkel, *Hardware-Software Cosynthesis for Microcontrollers*, IEEE Design&Test, Vol.10, No.4, December 1993.

[Buc94] J.Buck, S.Ha, A.Lee, D.G.Messerschmitt, *Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems*, Int. Journal of Computer Simulation, n.4, pp155-182, April 1994.

[Chi93a] 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.

[Chi94] M.Chiodo, P.Giusto, A.Jurecska, H.C.Hsieh, A.Sangiovanni-Vincentelli, L.Lavagno, *Hardware-Software Codesign of Embedded Systems*, IEEE Micro, Vol.14, n.4, August 94, pp.26-36.

[Cho94] P.Chou, E.A.Wlakup, G.Borriello, *Scheduling for Reactive Real-Time Systems*, IEEE Micro, Vol.14, n.4, August 94, pp.37-47.

[DeM90] G. De Micheli et al., *The Olympus Synthesis System*, IEEE Design and Test of Computers, Vol.7, N°5, October 1990, pp.37-53.

[For95] W.Fornaciari, A.Agostini, G.S.Sturniolo, N.Missere, M.Vincenzi, S.Prodi, *Hardware-Software Co-design within the TOSCA Design Environment*, in Proc. of IEEE-ICRAM95, Istanbul, Turkey, August 1995.

[Gaj94a] D.Gajski, F.Vahid, S.Narayan, *A System-Design Methodology: Executable-Specification Refinement*, in Proc. of EDAC'94, Paris, France, February, 1994.

[Gaj94b] S.Narayan, D.Gajski, *Synthesis of System-Level Bus Interfaces*, in Proc. of EDAC'94, Paris, France, February, 1994.

[Gup92] R.K.Gupta, C.Coelho, G.De Micheli, *Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components*, Proc. of the 29th DAC, June 1992.

[Gup93] R.K.Gupta, G.De Micheli, *Hardware-Software Cosynthesis for Digital Systems*, IEEE Design&Test, September 1993.

[Har87] D.Harel, *Statecharts: a Visual Formalism for Complex Systems*, Science of Computer Programming (1987), North-Holland.

[Har90] D.Harel et al., *STATEMATE: A Working Environment for the Development of Complex Reactive Systems*, IEEE Trans. on Software Engineering, Vol.16, No.4, April 1990.

[Hoa78] C. A. R. Hoare. *Communicating Sequential Processes*, Communications of the ACM, Vol. 18, No. 8, 66-77, agosto 1978.

[Ism94] T.Ismail, K.O'Brien, A.Jerraya, *Interactive System-level Partitioning with PARTIF*, in Proc. of EDAC'94, Paris, France, February, 1994.

[Jpb94] H. Jifeng, I. Page e J. Bowen, *Towards a Provably Correct Hardware Implementation of OCCAM*, Technical Report, Oxford University Computing Laboratory, 1994.

[Mok84] A.K.Mok, *The design of real-time programming systems based on process models*, in Proc. of IEEE Real-Time Systems Symposium, Dec 1984, pp 5-17.

[Stei93] U.Steinhausen, R.Camposano, H.Gunther, P.Ploger, M.Theibinger, H.Veit, H.T.Vierhaus., U.Westerholz, J.Wilberg, *System-Synthesis using Hardware/Software Co-design*, in Proc. of 2nd Workshop on HW/SW Co-Design, Cambridge, Massachussetts, October, 1993.

[Sut93] S.Sutarwala, P.Paulin, Y.Kumar, *Insulin: An Instruction Set Simulation Environment*, in Proc. of CHDL'93, pp355-362, Ottawa, Canada, April 1993.

[Vah95] F.Vahid, D.Gajski, *SLIF: A Specification-Level Intermediate Format for System Design*, in Proc. ED&TC 95, pp. 185-188.

[Vah95b] F.Vahid, D.D.Gajski, *Specification and Design of Embedded Hardware-Software Systems*, IEEE Design & test of Compuer, Spring 1995, pp. 53-67.

[Wol92] W.Wolf, A.Takach, C.Huang, R.Manno, *The Princeton University Behavioral Synthesis System*, 29th DAC, 1992.

[Wol94] W.H.Wolf, *Hardware-Software Co-design of Embedded Systems*, Proceedings of the IEEE, vol.82, n.7, July 1994.

Appendix 1.

The annotated VIS code for the Hamming decoder of fig.11 is here reported.

```
//PROC HammingEncoder (CHAN OF BYTE inChannel, CHAN OF [12]BIT
outChannel)
          <ports>
          inChannel
          outChannel
          </ports>

          <data>

//BYTE dataIn:
dataIn    defb          0

//[12]BIT dataOut:
dataOut   defw          0

//BIT b0,b1,b3,b7:
b0        defb          0
b1        defb          0
b3        defb          0
b7        defb          0

//BIT c2,c4,c5,c6,c8,c9,c10,c11:
c2        defb          0
c4        defb          0
c5        defb          0
c6        defb          0
c8        defb          0
c9        defb          0
c10       defb          0
c11       defb          0

          </data>

          <code>
//SEQ
```

```
//inChannel ? dataIn
```

```
          <process>
          <atomic><live none>                     //not necessary
          move.l        inChannel,R1 <2 2 6>
          call          read_byte   <2 4 6>       //result in R0
          move.b        R0,@dataIn(BP)            <2 4 4>    //save in R1
          </atomic>
          </process 6 10 16>
//PAR
          <group>

//c2 := dataIn[0]
          <process><use R0>
          move.b        @dataIn(BP),R0            <2 4 4>
          and.b         01h,R0       <1 2 4>//result in R0
          move.b        R0,@c2(BP) <2 4 4><free R0>
          </process 5 10 12>
```

.... similarly for c4, c5, c6, c8, c9, c110 ....

```
//c11 := data[7]
          <process><use R0>
          move.b        @dataIn(BP),R0            <2 4 4>
          and.b         80h,R0       <1 2 4>
          move.b        R0,@c11(BP)               <2 4 4><free R0>
          </process 5 10 12>
          </group>
//PAR
          <group>

//b0 := c2 BITOR c4 BITOR c6 BITOR c8 BITOR c10
          <process><use R0>
          move.b        @c2(BP),R0 <2 4 4>
          or.b          @c4(BP),R0 <2 4 4>
          or.b          @c6(BP),R0 <2 4 4>
          or.b          @c8(BP),R0 <2 4 4>
          or.b          @c10(BP),R0 <2 4 4>
          move.b        R0,@b0(BP) <2 4 4><free R0>
          </process 12 24 24>
```

... similarly for b1 and b3 ...

```
//b7 := c8 BITOR c9 BITOR c10 BITOR c11
          <process><use R0>
          move.b        @c8(BP),R0 <2 4 4>
          or.b          @c9(BP),R0 <2 4 4>
          or.b          @c10(BP),R0 <2 4 4>
          or.b          @c11(BP),R0 <2 4 4>
          move.b        R0,@b7(BP) <2 4 4><free R0>
          </process 10 20 20>

          </group>

//dataOut := [b0,b1,c2,b3,c4,c5,c6,b7,c8,c9,c10,c11]
          <process><use R0>
          move.b        @b0(BP),R0 <2 4 4>
          <use R1>
          move.w        0001h,R1     <2 2 6>
          and.w         R1,R0         <1 1 2>
          <use R2>
          move.b        @b1(BP),R2 <2 4 4>
          shl.w         #1,R1         <1 1 2>
          and.w         R1,R2         <1 1 2>
          or.w          R2,R0         <1 1 2><free R2>
          <use R2>
```

... similarly for c2, b3, c4, c5, c6, b7, c8, c9, c10 ...

```
          move.b        @c11(BP),R2 <2 4 4>
          shl.w         #1,R1         <1 1 2>
          and.w         R1,R2         <1 1 2><free R1>
          or.w          R2,R0         <1 1 2><free R2>
          <live R0></process 59 84 122>
//outChannel ! dataOut
          <process>
          <atomic><live R0>             //in this case atomic it is not
                                        //actually necessary
          move.l        outChannel,R1             <2 2 6>
          call          write_int     <2 4 6>     //dataOut is yet in R0
          </atomic>
          </process 4 6 12>
//:
          ret                                     <2 2 2><live none>
          </code>
```