CoWare – A design environment for heterogeneous hardware/software systems

Karl Van Rompaey, Diederik Verkest, Ivo Bolsens, Hugo De Man

IMEC, Kapeldreef 75, B-3001 Heverlee, Belgium
E-mail: {rompaey,verkest,bolsens,deman}@imec.be

Abstract

In this paper the design problems encountered when designing heterogeneous systems are studied and solutions to these problems are proposed. It will be shown why a single heterogeneous specification method ranging from concept to architecture is required and why it should cover issues as modularity, design for reuse, reuse of designs and reuse of design environments. A heterogeneous system design environment based on co-specification, co-simulation and co-synthesis is proposed and its application is illustrated by means of a spread spectrum based pager system.

1. Introduction

The design process for a DSP system must bridge the gap between the heterogeneous functional specification and its heterogeneous implementation. The shorter time to market and exponentially growing complexity of systems requires an increase of design productivity by at least one order of magnitude by the end of the decade. The goal of this contribution is to reflect on the impact of DSP systems and VLSI technology on design technology needs.

In section 2, we analyze the characteristics of DSP systems to derive initial requirements for a system design environment. Section 3 focuses on the design problems of heterogeneous system design. From these, additional constraints for the specification method are derived. A data model and language for the specification of heterogeneous systems is proposed in section 4. The effectiveness of this specification method is illustrated by a spread spectrum based pager example in section 5. Finally, in section 6, it is shown how the same data model can be used for modeling off-the-shelf processors. These models are used in processor independent interface synthesis tools.

2. Characteristics of heterogeneous systems

A typical example of a heterogeneous system, a spread spectrum based pager, is illustrated in Figure 1. Most DSP systems of this complexity, consist of one or more main DSP paths, slow control loops and a reactive control system taking events of a slow environment or slow status information from the DSP paths. The control loop and reactive control system set the mode or parameters of the DSP paths.

The main DSP path is usually a concatenation of data-flow components transforming the format of the data. When these data-flow components operate on unfragmented signal words they can best be specified as data-flow algorithms (e.g. in DFL), while others that manipulate individual bits of the signal can better be described as an FSM with datapath (e.g. in VHDL). The data-flow components in the DSP path, which can operate at fairly different data- and execution rates, are internally strongly connected data-flow graphs with sparse external communication. Hence, from an implementation point of view they are seldom partitioned over several hardware components. Rather they will be merged.

The slow synchronization loops and mode control, which run in parallel with the data-flow, have reactive semantics and can best be described with a Program State Machine model [5] which is a hierarchy of program-states, in which each program state represents a distinct mode of computation. The slow nature of the synchronization and mode control make them natural candidates for software implementation (e.g. in C). The same holds for the low data-rate parts of the system (frame extraction, channel decoding).

From the above it follows that at the conceptual level DSP systems need a lot more than data-flow models (SDF, DDF) for their complete specification. Also control-flow oriented and reactive models are required. Hence, there is a need for a co-specification environment that allows the system components to be described with the most appropriate host language. Next we will take a look at the problems involved in implementing heterogeneous systems.

3. Heterogeneous system design problems

Today the conceptual specifications used by system designers are barely understood by chip architects, thinking in VHDL terms (if already). Hence most specifications are first translated into (non-executable and ambiguous) English and

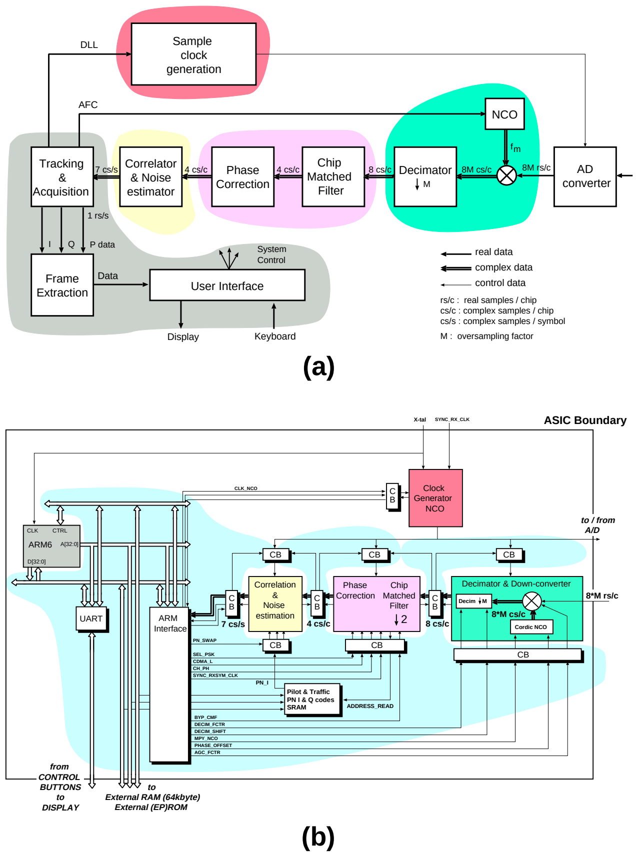


Figure 1. Functional specification (a) and possible implementation (b) of a pager.

then virtually redesigned in VHDL for the hardware components and C or assembly for the software components. Although the hardware and the software components have a tight interaction, *both hardware and software are designed separately*. Only after system assembly, the software is run on the hardware. As a consequence, the hardware realizing the hardware/software interface can be erroneous or far from optimal, requiring a redesign of the system.

Another problem is the ever increasing complexity of systems, which is coped with by designers by means of reuse, modularity and design abstraction.

Reuse: What was a complex system in the past is used as an abstract and reusable component in future systems (ALUs are used in an ARM processor core, which in turn is used as a component in a pager system). In most design environments, reusable components are characterized according to a model and stored in a library. In heterogeneous system design, the architectures consist of communicating processors: reuse happens at the processor level. Therefore, there is a need for a processor model that allows to model off-the-shelf processors on an as-is basis to support a "Reuse of Design" methodology.

The main difficulty in reusing off-the-shelf processors lies in the fixed communication protocols they use. When processors with incompatible protocols have to be interfaced, protocol conversions are required. A good selection of the protocol is possible only when all processors involved in the communication are known because it is influenced by a number of factors such as the required speed and robustness of the data transfer, or the relation between the clocks of the processors. Therefore, we advocate a "Design for Reuse" methodology which supports a strict separation between functional and communication behavior. Initially, a component is described purely functional. Later, when the component is used in the system, the design environment must allow to plug in the most appropriate communication behavior. This approach is in contrast with current hardware (VHDL) design practices, where communication and functional behavior are mixed.

Modularity: In modular designs, the complete system functionality is split into communicating components of manageable complexity. The advantage of this approach is that the components can be reused and that the system is easier to adapt and maintain. The disadvantage is the overhead because of the inter-component communication or because the compiler does not optimize over the component boundaries. Therefore, the inter-component communication semantics should be such that modularity can be removed easily when merging two components into a single component.

Design Abstraction: In the past, a lot of effort has been put in design environments that allow to implement today's system components. Languages with associated simulators, tuned towards specific application domains, allow to specify and simulate components at a high abstraction level. Hardware compilers can implement the component description onto processors with highly specialized architectures. Software compilers allow to generate machine code for off-the-shelf programmable processors. Instruction set simulators allow to debug the machine code at different levels of abstraction. However, what is missing is the glue that links these design environments together and automatically interfaces the generated or off-the-shelf processors according to the system specification. Hence, a system design environment should allow to include existing design environments easily and should provide synthesis tools for hardware/hardware and hardware/software interfacing that are processor and design environment independent.

The gap between the conceptual system specification and the system implementation is rapidly becoming the most important bottleneck of the design process. There is a need for a formal and executable specification hand-over from system designer to system implementor and an environment that allows to gradually refine the system designer's specification to an architecture, which serves as a specification understood by the chip architects (hardware: VHDL, software: C). Processor independent hardware/hardware and hardware/software interfacing are key issues. Simulation is required after every refinement step and at any abstraction level. To cope with complexity, system level simulations at different abstraction levels and mixed software/hardware simulations are essential. Off-the-shelf components and their design environments must be reused and in-house components should be designed for reuse.

4. Heterogeneous system specification method

From the above requirements, a data model and language has been developed on which the CoWare system design environment is built. In the data model, a lot of attention has been paid to the inter-component communication semantics, which is the key to effective solutions for modularity, "Design for Reuse", "Reuse of Designs", communication refinement and interface synthesis.

Modularity in the specification is provided by means of processes. Processes contain language encapsulations[1] which are used to describe the behavior of a system component. Communication between processes takes place through a behavioral interface, consisting of ports. For two processes to be able to communicate, their ports must be connected with a channel. The communication semantics are based on the concept of the Remote Procedure Call (RPC), i.e. one process can trigger the execution of a thread in another process. The semantics of the data model objects is summarized below.

A single *process* can have multiple host language *encapsulations* describing different implementations for the same component, or for the same component represented at different abstraction levels. For example the Chip Matched Filter in the pager example of Figure 1 was initially described by means of a DFL encapsulation (conceptual level). During refinement, this DFL encapsulation was compiled by the Cathedral compiler [10] into a VHDL encapsulation, which is the processor implementation of the Chip Matched Filter behavior (structural-level). If a process has multiple encapsulations, then these should all be functionally equivalent. Encapsulations can be primitive or hierarchical. A hierarchical encapsulation is one described with the CoWare language. In a CoWare encapsulation one can instantiate processes and connect their ports with channels. All other encapsulations are primitive and consist of a context and a number of threads. The *context* contains code that is common to all threads in the encapsulation, i.e. variables/signals and functions as allowed by the semantics of the host language. As such the context provides for inter-thread (intra-process) communication.

[1] Currently C, DFL, VHDL and CoWare are supported.

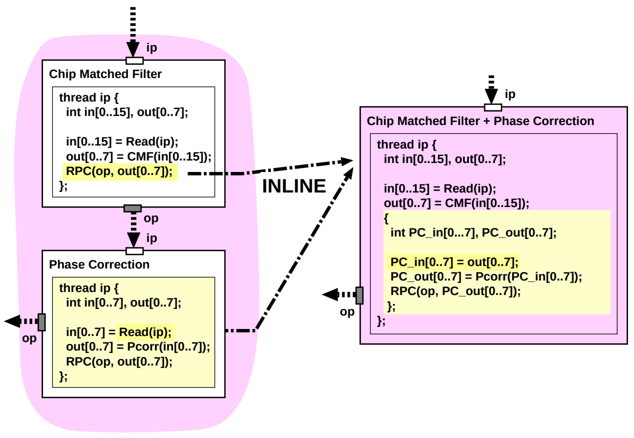
Ports are objects through which processes communicate. Ports can be primitive or hierarchical. Hierarchical ports are used to describe protocol conversions and data formatting processes. A primitive port is used in all other cases. It consists of a protocol and a data type parameter.

Protocols define the communication semantics of a port. Protocols can be primitive or hierarchical. Each primitive protocol indicates another way of data transport and is characterized by a data direction (in, out, or inout) and a control direction (master, slave). The control direction indicates whether the protocol activates an RPC (master) or services an RPC (slave). In the remainder of this text, ports with a slave/master protocol are also referred to as slave/master ports. Hierarchical protocols refine the primitive protocol with a timing diagram and the associated I/O *terminals*.

A *thread* is a single flow of control within a process. A process can contain multiple threads. We distinguish between *slave threads* and *autonomous threads*. Slave threads are uniquely associated to slave ports and their code is executed when the slave port is activated. Autonomous threads are not associated to any port and their code is executed, after system initialization, in an infinite time-loop.

A *channel* is point-to-point. Two ports that are connected by a channel can exchange data. A channel can be primitive or hierarchical. A *primitive channel* provides for unbuffered communication. It has no behavior: it is a medium for data transport. In hardware it is implemented with wires, in software it is implemented with a function call. A *hierarchical channel* refines a primitive channel by specifying a behavior. At the conceptual level, a hierarchical channel can be used to model a communication channel, e.g. to model bandwidth limitations. At the implementation level, a hierarchical channel is used to specify a communication buffer, e.g. a FIFO.

The CoWare data model supports three *communication mechanisms*. Communication always happens between two threads. If the threads are part of the same process, we speak about intra-process communication. If the threads are part of different processes, we speak about inter-process communication in which case we make a further distinction based on the protocol. *Intra-process communication* is done by making use of shared variables/signals that are declared in the context of the process. Avoiding that two threads access the same variable at the same time is host language dependent. It is the user's responsibility to protect critical sections using the mechanisms provided in the host language. *Inter-process communication with a primitive protocol* is RPC based. On a master port the RPC function can be used to initiate a thread in a remote process. The RPC function returns when the slave thread has completed. In the slave thread



```
thread ip {
  int in[0..15], out[0..7];

  in[0..15] = Read(ip);
  out[0..7] = CMF(in[0..15]);
  RPC(op, out[0..7]);
};
```

```
thread ip {
  int in[0..7], out[0..7];

  in[0..7] = Read(ip);
  out[0..7] = Pcorr(in[0..7]);
  RPC(op, out[0..7]);
};
```

```
thread ip {
  int in[0..15], out[0..7];

  in[0..15] = Read(ip);
  out[0..7] = CMF(in[0..15]);
  {
   int PC_in[0...7], PC_out[0..7];

   PC_in[0..7] = out[0..7];
   PC_out[0..7] = Pcorr(PC_in[0..7]);
   RPC(op, PC_out[0..7]);
  };
};
```

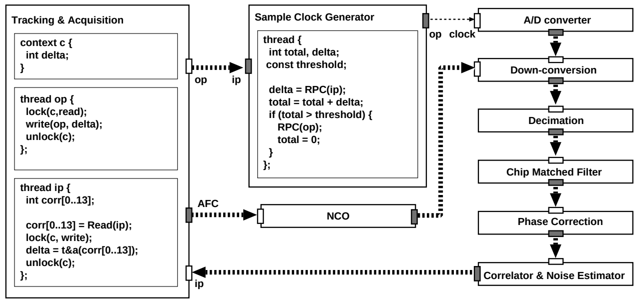
Figure 2. Merging of processes.

(uniquely associated with a slave port), the Read and Write functions can be used to access the slave port. *Inter-process communication with a hierarchical protocol* is completely defined by the timing diagram and terminals of the protocol.

The protocol hierarchy provides a clear separation between functional and communication behavior. Initially, a component is described purely functional. In such a description one can perform abstract actions on a port. The allowed actions are determined by the primitive protocol of the port. In addition, one can attach a behavior, also called slave thread, to a slave port. Later, when the component is instantiated in a system, the primitive protocol is refined into the best suited hierarchical protocol, taking into account the other system components. This fixes the timing diagram and terminals used to communicate over that port. Next, the port containing the hierarchical protocol is made hierarchical to add the required communication behavior that implements the timing diagram of the selected hierarchical protocol. This method reduces the amount of protocol conversions needed at the system level and allows to postpone the selection of the communication protocol and its implementation until late in the design process, in this way achieving the requirements of "Design for Reuse". The concept of hierarchical protocols is also useful to model off-the-shelf components ("Reuse of Design"), because the timing diagrams according to which a processor communicates are abstracted in it.

Due to the selection of RPC as inter-process communication, the classification of protocols and the structuring of a process in encapsulations with context and threads, an *inlining transformation* can be implemented that allows to remove modularity. The transformation allows to merge processes into a single process[2]. In the process of merging, all RPC calls are in-lined: each slave thread is in-lined in the

[2] Merging of processes is only allowed when the processes have an encapsulation in the same host language.



```
context c {
  int delta;
}
```

```
thread op {
  lock(c,read);
  write(op, delta);
  unlock(c);
};
```

```
thread ip {
  int corr[0..13];

  corr[0..13] = Read(ip);
  lock(c, write);
  delta = t&a(corr[0..13]);
  unlock(c);
};
```

```
thread {
  int total, delta;
 const threshold;

  delta = RPC(ip);
  total = total + delta;
  if (total > threshold) {
     RPC(op);
     total = 0;
  }
};
```

Figure 3. The pager described with RPC.

code that calls it via an RPC statement. This in-lining transformation eliminates the overhead that accompanies remote procedure calls. It reduces the number of threads and therefore the overhead that accompanies the switching between threads. Finally, it allows compilers to optimize over the boundaries of the original processes. Figure 2 shows the effect of merging the Chip Matched Filter process and the Phase Correction process of Figure 1.a. On the right-hand side the RPC call in the Chip Matched Filter process has been replaced with the code of the slave thread in the Phase Correction process.

5. The design of a pager

In this section, the data model explained above will be applied to the design of a spread spectrum based pager.

Conceptual specification. Figure 1.a shows how the pager was described in CoWare. Each component corresponds to a process implementing a specific function of the pager. This functional decomposition determines the initial partitioning. The arrows in between the processes indicate communication via a Remote Procedure Call (RPC) mechanism. Figure 3 shows the RPC communication in detail for part of the pager design. The blocks in the figure correspond to the processes from Figure 1.a. The small rectangles on the perimeter of the processes are the ports. The shaded ports are master ports, the others are slave ports.

The Sample Clock Generator process contains an autonomous thread that performs an RPC over its input port ip to the Tracking & Acquisition process to obtain a new value for delta, which is added to some internal variable until a threshold is exceeded, at which point an RPC call is issued to the A/D converter process. In this way the Sample Clock Generator process implements a sawtooth function of which the period is influenced by the value of delta. In the Tracking & Acquisition process, the slave thread op services RPC calls over the port op by writing the value of the variable delta to the port. The variable delta is declared in the context of the Tracking & Acquisition process and is updated by another slave thread ip, which is called from the Correlator & Noise Estimator process.

This example shows how the context is used for communication between threads inside the same process whereas the RPC mechanism is used for communication between threads in different processes.

Figure 1.b shows the architecture description of the pager. Architectures are again specified as an interconnection of processes. The processes, representing processor implementations, are described with a structural VHDL encapsulation and have ports with a hierarchical protocol. The architectural description is the result of a number of refinement steps.

Partitioning and mapping. In a first step, partitioning and mapping, as indicated in Figure 1, takes place. The NCO, Down-conversion, and Decimation processes, for example, are merged and mapped in hardware onto an application specific DSP Cathedral processor [10] mainly because the sample rate of the merged processes is identical, which implies that they can be clocked at the same frequency. The Tracking & Acquisition, Frame Extraction, and User Interface processes, for example, are merged and mapped on a programmable processor. For this design an ARM6 processor is chosen. To obtain a maximal degree of flexibility as much of the functionality as possible is implemented in software on the ARM6. The Tracking & Acquisition process has to be implemented in software because the algorithm used to perform tracking and acquisition may require modification depending on the application domain of the pager system. The Correlator & Noise Estimator process is not included in software because the input rate for the Correlator & Noise Estimator is too high to realize a real-time communication between the ARM6 and the Phase Correction process. In addition an estimation of the number of cycles required to execute each function on the ARM6 shows that the implementation of the Correlator & Noise Estimator process in software leaves insufficient time to perform tracking and acquisition in between every two symbols.

Communication selection. After partitioning and mapping, communication selection is performed. In the architecture of Figure 1.b the processors can, in principle, operate concurrently because each processor has its own thread of control. This can be achieved by refining the RPC based communication scheme to pipeline the processors: all processors operate concurrently and at I/O points they synchronize. In the CoWare design environment the refinement of the communication mechanism is performed by making use of a *hierarchical channel*. Hierarchical channels, the CB components in Figure 1.b, replace a primitive channel by a process that describes *how* communication over that channel

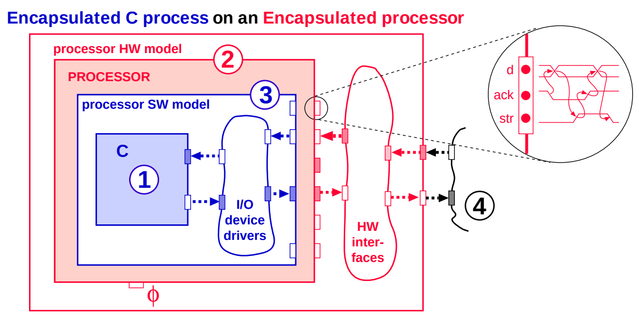


Figure 4. Mapping a C process on a programmable processor.

is carried out.

Implementation of components. The next step is to compile the merged components into an implementation.

The merged User Interface, Tracking & Acquisition and Frame Extraction process is implemented in software on an ARM core by means of the tool SYMPHONY which implements a (software) process on a programmable processor and connects it with the (hardware) processes it communicates with. This task is schematically depicted in Figure 4. SYMPHONY makes use of a library of processor models and I/O scenario models both described using the CoWare data model, to generate the I/O device drivers and the hardware interfaces. The I/O device drivers link the original C process (1) to the software interface of the processor (3). The hardware interface makes the link between the hardware interface of the processor (2) and the original processes (4) external to the processor.

For the CB components an encapsulation with a gate-level VHDL description is selected from a library.

All other components are compiled with the Cathedral silicon compiler [10] into a VHDL encapsulation.

Interface synthesis. Finally, all processors have to be interfaced: i.e. their protocols have to be made compatible. This is done by INTEGRAL [8], a processor independent, hardware/hardware interfacing tool. The tool takes two timing diagrams and a protocol independent description of the communication behavior and generates a gate-level VHDL description of a circuit that interfaces the two timing diagrams.

6. Modeling off-the-shelf processors

In the architecture of a heterogeneous system off-the-shelf processors, such as the ARM core, are used. These processors come with their own simulation and compilation environment. For example the ARM core comes with a C-compiler and an instruction set simulator (ARMulator). After integration of the ARM design environment, it must be possible to compile a component described in C with the ARM C-compiler and simulate it with the ARMulator. As a consequence, plugging in new compilers and simulators into the system design environment must be easy and should not require new tools to be written. Furthermore, it must be possible to interface the off-the-shelf processor to the other system components. For that purpose, architecture and processor independent hardware/hardware and hardware/software interface tools are required. To build such tools, processor models are required. These models abstract the design information needed to perform interface synthesis and allow to reuse the processor in different system contexts.

Programmable processors require both hardware and software interfacing, and hence are abstracted in a hardware model and a software model. The processor *hardware model* formalizes the information that is available in the hardware section of the data sheet. In Figure 4, this hardware model is represented by the rectangle marked "processor HW model" and the ports at the outside of the rectangle's perimeter. All ports have hierarchical protocols: they consist of terminals and a timing diagram (including timing constraints) as is shown in Figure 4 for one particular port. The processor *software model* formalizes the information that is available in the software section of the data sheet. In Figure 4, this software model is represented by the rectangle marked "processor SW model" and the ports at the inside of the rectangle's perimeter. The software model identifies, for example, what ports can be used as interrupt ports and what their characteristics are (priority of the interrupt, maskable interrupt or not, ...). For the ARM processor, the memory and co-processor interfaces are modeled as bidirectional (inout) master ports, and the irq and fiq ports are modeled as slave ports. The software model also contains a behavioral description that contains all code for processor specific actions such as installing interrupt vectors, or reading and writing to specific memory locations via the memory port.

7. Conclusions

To date, some system design environments include hardware/software co-design and support heterogeneous system implementation (programmable processors combined with custom and off-the-shelf hardware components). However, they are based on a single specification paradigm often combined with a single associated specification language [2, 3, 6, 7, 9]. The specification paradigm reflects the target application domain which usually is control-oriented. The benefit of such an approach is in its high analytic potential that allows automation of design tasks such as hardware/software partitioning, (formal) analysis of timing properties, or the optimization of the design with respect to timing constraints.

True *heterogeneous* system design environments [1, 4], in contrast to the above approaches, start from a heterogeneous specification in which different paradigms and languages can be combined and, as such, are not application domain oriented. The benefit of this approach, at the expense of analytic power, is the ability to arbitrarily couple simulation paradigms and tools, allowing a designer full freedom in expressing each part of the system in the most appropriate language. Both ESCAPE and Ptolemy are oriented to heterogeneous system simulation combining different simulation paradigms. Unlike ESCAPE and Ptolemy, the environment presented in this paper is implementation oriented.

The CoWare environment supports the design of heterogeneous hardware/software systems. It allows to model systems at the conceptual level, at the architectural level, and allows to represent the refinement process from specification to implementation. The CoWare environment allows to abstract off-the-shelf processor through the use of processor models, and allows the automatic generation of hardware/hardware and hardware/software interfaces, based on these processor models.

Acknowledgments. We gratefully acknowledge the financial support for this research of ESTEC under the SCADES-2 and SCADES-3 projects, and of OMI under the Standards-2 project.

References

[1] J. Buck et al. PTOLEMY: A framework for simulating and prototyping heterogeneous systems. *International Journal on Computer Simulation*, Jan 1994.

[2] P. Chou et al. The Chinook hardware/software co-synthesis system. *Proc. ISSS 95*, pp. 22-27. Cannes, Fr, Sep 1995.

[3] R. Ernst et al. Hardware-software cosynthesis for microcontrollers. *IEEE Design & Test*, pp. 64-75, Dec 1993.

[4] H. Fleurkens et al. ESCAPE: A flexible design and simulation environment. *Proc. SASIMI 93*. Nara, Japan, Oct 1993.

[5] D. Gajski et al. *Specification and Design of Embedded Systems*. Prentice-Hall, 1994.

[6] R. Gupta et al. Hardware-software cosynthesis for digital systems. *IEEE Design & Test*, pp. 29-41, Sep 1993.

[7] A. Jerraya et al. SOLAR: An intermediate format for system-level modeling and synthesis. In J. Rozenblit and K. Buchenrieder, editors, *Computer Aided Software/Hardware Engineering*, IEEE Press, 1994.

[8] B. Lin et al. Synthesis of concurrent system interface modules with automatic protocol conversion generation. *Proc. ICCAD 94, pp. 101-108, San Jose, CA*, Nov 1994.

[9] S. Narayan et al. System specification with the SpecCharts language. *IEEE Design & Test*, pp. 6-13, Dec 1992.

[10] J. Vanhoof et al. *High-Level Synthesis for Real-Time Digital Signal Processing.* Kluwer Academic Publ., Boston, 1993.