

# Hardware/Software Co-Design of Digital Telecommunication Systems

IVO BOLSENS, HUGO J. DE MAN, FELLOW, IEEE, BILL LIN, KARL VAN ROMPAEY, STEVEN VERCAUTEREN, AND DIEDERIK VERKEST

## *Invited Paper*

*In this paper we reflect on the nature of digital telecommunication systems. We argue that these systems require, by nature, a heterogeneous specification and an implementation with heterogeneous architectural styles. CoWare is a hardware/software co-design environment based on a data model that allows to specify, simulate, and synthesize heterogeneous hardware/software architectures from a heterogeneous specification. CoWare is based on the principle of encapsulation of existing hardware and software compilers and special attention is paid to the interactive synthesis of hardware/software and hardware/hardware interfaces. The principles of CoWare will be illustrated by the design process of a spread-spectrum receiver for a pager system.*

## I. INTRODUCTION

Digital communication techniques form the basis of the rapid breakthrough of modern consumer electronics (CD, CD-I, DCC, DAB, ...), wireless and wired voice and data networking (ISDN, GSM, DECT, MSBN, Videophone, ...), broadband networks (ATM, ADSL, ...) and multimedia. All these products belong to the fastest growing industrial activity on earth today. Digital communication systems are made possible by the combination of very large scale integrated (VLSI) technology and digital signal processing (DSP). DSP systems perform real-time mathematical transformations on time discrete digitized samples of analog quantities with finite bandwidth and signal-to-noise ratio (SNR). These transformations can be specified in programming languages and executed on a programmable processor or they can be implemented on application specific hardware. The choice is entirely determined by trade-offs between cost, performance, power, and flexibility. Hence DSP is a candidate par excellence for hardware/software co-design. DSP based products have a growth rate of 38%

Manuscript received February 1, 1996; revised December 2, 1996. This work was supported in part by ESA under the SCADES-2 and SCADES-3 projects, and in part by the EC's OMI Standards-2 Project.

I. Bolsens, B. Lin, S. Vercauteren, and D. Verkest are with IMEC, B-3001 Leuven, Belgium (e-mail: bolsens@imec.be and verkest@imec.be).

H. J. De Man is with the Katholieke Universiteit Leuven, B-3001 Leuven, Belgium.

K. Van Rompaey is with CoWare N.V. B-3001 Leuven, Belgium.  
Publisher Item Identifier S 0018-9219(97)02052-5.

per year. The average time window for a new product is as short as 18 months. Hence a very high design productivity is required. This design productivity is jeopardized by the growing gap between system designers and chip architects.

- Communication systems are conceived by highly specialized system design teams presently designing at the board level and thinking in terms of executable concurrent programming paradigms which are not well understood by chip architects today. Most specifications are first translated into English and then redesigned in VHDL by chip architects. This gap between system design and implementation is rapidly becoming the most important bottleneck in the design process. Future CAD systems must make a seamless handover possible from system designer to chip architect.
- Silicon complexity in 2000 will require design at the rate of a "16-bit processor a person day." Chips will have to be designed at the processor-memory level, not at the gate or RT-level. This requires reuse of designs as well as a design for reuse methodology and implies hardware/software co-design at the chip level. Hence, silicon designers have to move up another level of abstraction.
- On the other hand, in spite of faster time to market and exponentially growing complexity, the size of the design teams does not seem to grow.

All the above leads to a need to increase design productivity by at least one order of magnitude by the end of the decade. Clearly this requires a design technology that enables a seamless transition from a software centric system specification to an implementation in reusable hardware/software components at the processor-memory level. This is the subject of this paper.

In Section II, we first analyze the characteristics of specifications and architectures of the DSP-oriented systems that form the basis of digital communication systems. This is followed by a discussion of the design process of such systems. In Section III this leads to the definition of a

data model that can be used all the way from a heterogeneous specification to a heterogeneous implementation. Section IV focuses on synthesis methods for interprocessor communication, as this will be identified as the key design problem at the processor-memory level. In Section V, we present the CoWare view on co-simulation. The CoWare data model and the co-design environment built on top of it will be illustrated in Section VI by a practical hardware/software co-design example of a satellite pager.

## II. SPECIFICATION, ARCHITECTURE, AND DESIGN PROCESS FOR DIGITAL COMMUNICATION SYSTEMS

In this section we first focus on the specification of the DSP parts of digital communication systems. We will then extend that to a typical communication system that comprises besides DSP, control loops, and UI parts. Such specifications consist of communicating heterogeneous processes. We then show that the implementation of such systems consists of very heterogeneous components and the communication between them. The art of design of these systems is then a gradual, library-based refinement of heterogeneous specifications into a heterogeneous architecture.

### A. The Specification View of DSP Systems

A digital signal results from a binary encoding of time and range discretized measurable continuous time, continuous range quantities. Sampling occurs at or above Nyquist frequency and coding is done with just enough word length to satisfy the SNR. Hence, in contrast to general purpose computing, digital signals are usually of the fixed-point type as this leads to hardware and power savings for the required computational performance.

Due to the periodic sampling, digital signals behave as a stream of digital words. In addition, the digital words are naturally structured into multidimensional arrays, which are to be processed within a frame period  $T_f$  [1]. The *frame period*  $T_f$  is a periodic hard real-time constraint imposed by the bandwidth of the signal.

In the most strict sense DSP systems are algorithms mapping digital signals into digital signals. The real time constraint is determined by the frame period  $T_f$  which is also the period of the algorithm for consuming an array of input samples and producing a new output. The periodicity of this constraint and the nature of the signals leads to the fact that the elementary DSP algorithm is a dataflow function [2]–[4].

A synchronous dataflow (SDF) algorithm [3], [4] can be modeled as an acyclic graph where nodes are combinational arithmetic operators and edges represent data precedences. This graph is called a dataflow graph (DFG). An operator can execute when a predetermined, fixed number of data (tokens) are present at its input and then it produces a fixed number of output data (tokens). Conditional selection of two operations to a single output is allowed. Operators have no state but arrays resulting from one algorithm execution can be saved for reuse in future executions (delay operator).

Many DSP algorithms are of this type. They can be described very efficiently by so called applicative program-

ming languages like SILAGE [5], DFL [6], and LUSTRE [7]. The advantage of such languages is that operations can be scheduled at compile time and execution of compiled code can be two orders of magnitude faster than execution of event driven code such as in, e.g., VHDL simulation.

In contrast, dynamic dataflow (DDF) algorithms [2] contain data-dependent token production and consumption. They allow for while and if-then-else constructs. For a detailed treatment on how conditions can be checked to statically schedule DDF, see [2]. In many practical DSP systems, hard periodic constraints are satisfied by inserting exit paths in while loops that guarantee a bounded execution time whereby unassigned signals are assigned default values. One can then transform these DDF's into a worst case SDF and schedule it statically. Otherwise, DDF's can be partitioned into a set of compile time scheduled SDF functions fired by internal or external Boolean events. One must then schedule these SDF blocks at run-time within the input-output (I/O) timing constraints of the DSP signals and external events. In case of hardware implementation, this requires the synthesis of a run-time control shell around the implementations of the SDF blocks. In case of software implementation on programmable DSP processors one can use the concept of real-time software synthesis [8]–[10]. The run-time scheduling, based on the occurrence of the events makes use of all the static information and is more efficient than using a real-time operating system (RTOS).

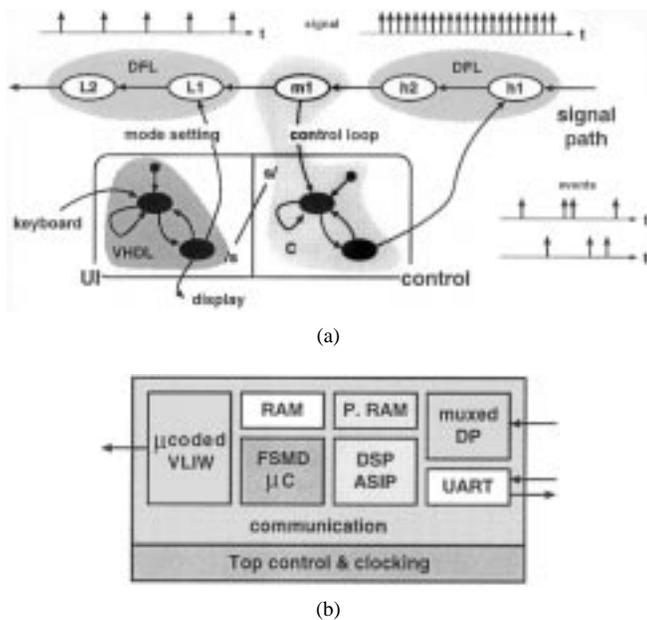
CAD systems for DSP such as DSP-Station of Mentor Graphics [11], PTOLEMY [12], GRAPE-II [13], COSSAP from Synopsys, and SPW from The Alta [14] Group all allow for specification of SDF and DDF and use as much as possible static scheduling in order to provide simulation speeds that are orders of magnitude faster than event driven simulators such as in use for VHDL. This justifies the use of these simulation paradigms for DSP specification and validation.

However, when we consider digital communication systems, a wider scope is necessary as illustrated in Fig. 1 which is an abstraction of many practical implementations of DSP systems and of which Fig. 2, to be discussed later, is a good practical example.

A careful look at Fig. 1 allows us to identify the common characteristics of system specifications.

- 1) Communication systems typically consist of one (or more) signal paths ( $h1, h2, \dots, L2$ ) as well as slow control loops and a reactive control system. The latter two observe events of a slow environment, such as commands from the user interface (UI) and slow status information of the signal paths, and control the mode of operation of the signal paths or send output data to the UI.
- 2) A *signal path* is usually a concatenation<sup>1</sup> of dataflow functional blocks (DFFB) often operating at fairly different data and execution rates. The rate differences

<sup>1</sup>Loops in a main signal path can only occur if they are cut by at least one algorithmic delay. The DFG then becomes acyclic again and is SDF if its constituting blocks are SDF.



**Fig. 1.** Heterogeneous functional specification (a) leads to heterogeneous implementation (b) at processor-memory communication level. System implementation requires refinement of (a) to (b).

naturally result from the typical structure of any digital communication system [15]. A typical receive path, such as in Figs. 1(a) and 2(a), may for example contain high speed down-converters, matched filters, and correlators ( $h1, h2$ ), followed by a low rate demodulator  $m1$  (symbol to bits format translation), and error correction in the base band signal  $L1$  (Reed–Solomon), and speech decompression  $L2$  (rate conversion). When the DFFB's operate on unfragmented signal words, they can best be specified as dataflow algorithms. When the DFFB's manipulate individual bits of the signals (e.g., modulation and Viterbi channel encoding), they can be directly specified as finite state machine with datapath (FSMD) processors [16]. Hence, the specification paradigm depends on the DFFB type.

- 3) DFFB's in the signal path are internally strongly interconnected DFG's with sparse external communication. Hence, from an implementation viewpoint, they are seldom partitioned over several hardware or software components. Rather, as shown in Figs. 1(a) and 2(a), they will be merged onto the same component if throughput and rate constraints allow to do so. Merging implies synchronizing and sometimes sequentializing the concurrent processes on a single component while still satisfying the timing constraints.
- 4) Control loops and mode control by parameter setting are common to all communication systems. Typical are the tracking and acquisition loops in order to synchronize frequency and phase of the receiver signal path to the characteristics of the incoming signal. Design of these loops constitutes one of the most difficult aspects of such systems since their characteristics depend strongly on noise and distortion in the communication channel. Many times it

requires the design of PLL's, DLL's, and FFT's controlled by "events" disturbing the regularity of the signal streams. The occurrence rate of these events is many orders of magnitude slower than the data rate in the signal path itself. Hence, similar to the UI, the processes modeling these slow control loops and mode setting have no dataflow but reactive semantics. They run concurrently with the dataflow and consist themselves often of concurrent processes.

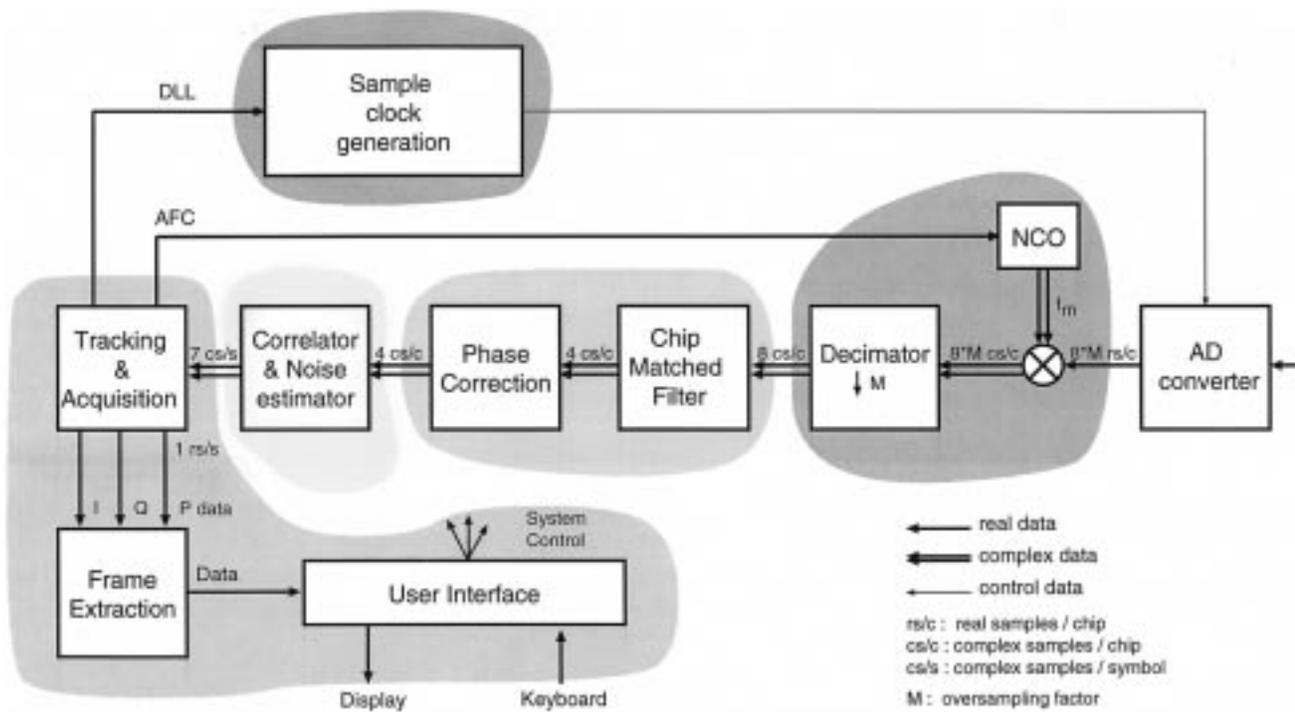
Such a control dominated system can be described as a *Program State Machine* (PSM) [16] which is a hierarchy of program-states, in which each program-state represents a distinct mode of computation. In [16] it is shown that VHDL is too low level to describe PSM's. Formalisms such as StateCharts [17] or SpecCharts [18], which include behavioral hierarchy, exception handling, and interprocess communication modeling [19], are needed to describe such systems. In practice, synchronization is often specified in one or more (concurrent) C programs.

In as far as the UI is concerned, it can also be described by a relatively slow PSM formalism as it behaves quite similar to the control loop behavior in that it sets the mode of operation of the DFFB's dependent on the user commands.

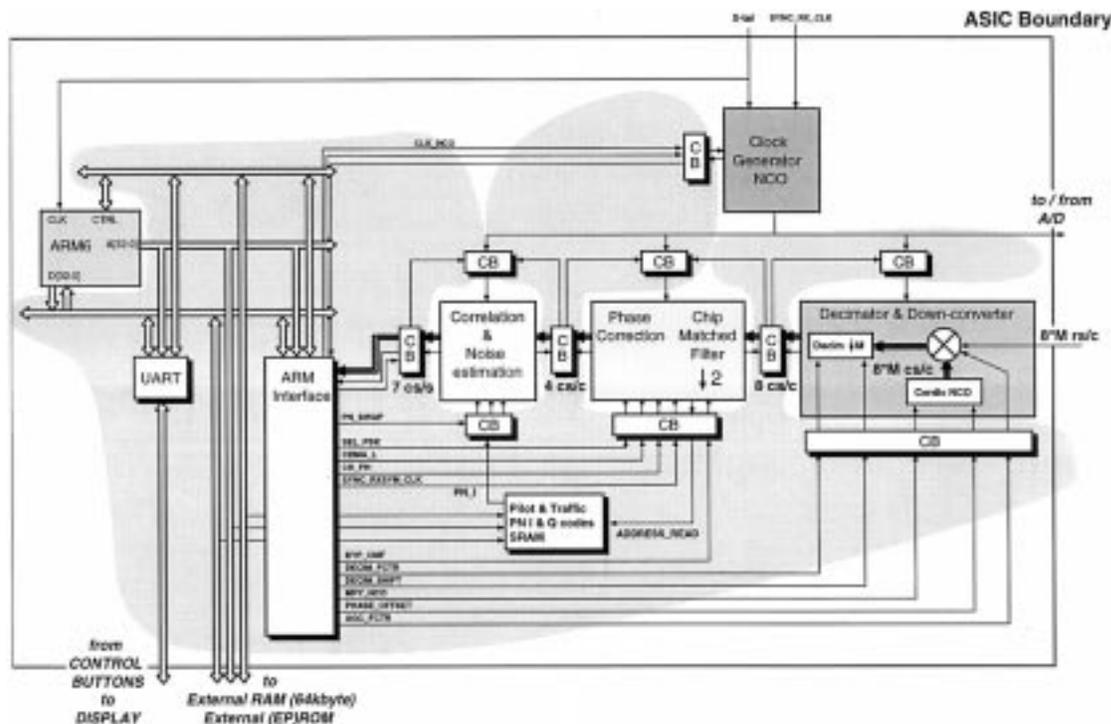
System design thus requires the ability to mix dataflow and reactive paradigms with widely different time constants. This time tyranny problem poses special problems in simulation and requires that all processes must be simulatable at the highest possible abstraction level, i.e., efficiently co-simulate dataflow and event-driven models. Especially in the case of video and image processing there is virtually no other way for validation than emulation. Rapid prototyping, allowing for close to real-time operation, is a bare necessity for such systems [20], [21]. This leads to the need for retargetable synthesis techniques that allow, on the one end, to map onto architectures of FPGA's, programmable DSP's and video signal processors (VSP's) and, on the other end, to map onto the final on-chip architecture to be discussed in Section II-B. We will show in Section III that CoWare provides a data model and a programming plug-and-play environment to do this.

From the above it becomes clear that the specification of communication systems requires more than one single paradigm. In Section II-B we will show that also the implementation of these systems consists of a heterogeneous architecture [22], [23]. This architecture can be a network of workstations for simulation, a network of FPGA's and programmable processors for rapid prototyping or a multiprocessor network on a final chip implementation.

Section III describes PTOLEMY [12] and CoWare [24] as examples of design environments that cope with the heterogeneity of system specification. CoWare also offers a programming environment to refine the specification all the way down to implementation. Key to the above will be the encapsulation of existing component compilers and library based synthesis of the communication between the different components.



(a)



(b)

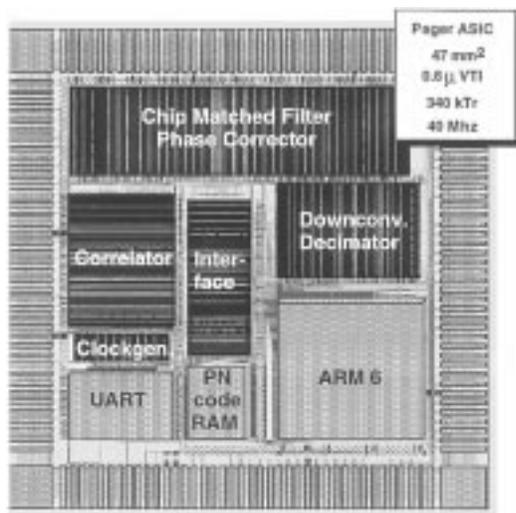
**Fig. 2.** (a) Functional specification of a spread-spectrum pager and (b) implementation using an ARM processor-accelerator memory architecture.

### B. System Architectures and the Design Process

The art of design consists of mapping a functional specification, as shown in Figs. 1(a) and 2(a) into a communication between several hardware components, as shown in Figs. 1(b) and 2(b). The hardware components in the architecture can be of a very different nature depending on performance, area, power, and reuse considerations.

Below we discuss the components encountered on digital communication chips and relate them to the specification paradigms discussed above.

1) *Implementation of the Signal Path:* Signal paths are characterized by concatenations of DFFB's consisting of nested loops with manifest loop bounds and iterating over multidimensional arrays to be executed periodically every



(c)

Fig. 2. (Continued.) (c) Layout of the pager.

time frame  $T_f$ . The nodes in the DFG in the DFFB's represent predominantly arithmetic register transfer (RT) operations on Boolean arrays. The area cost of an operator implementing an arithmetic operation is usually larger than the area cost of a multiplexer. Hence area optimization is characterized by the number of operations that can be time multiplexed on the same operator.

In [25] it is shown that the multiplexing degree has a strong impact on the architectural style suited to implement a given algorithm. On the other hand the occurrence of widely different data rates in the same system makes that the multiplexing degree can vary from less than ten to a few hundred thousand leading to a large heterogeneity in architectural styles on the same chip.

In base band voice or audio systems or back-end image processing, typically a few hundred scalar samples are processed in a time frame of the order of 10 ms. We call such systems *low throughput* systems [25]. Low throughput algorithms typically consist of a concatenation of many small irregular flowgraphs with little topological similarity. They contrast with *high throughput* systems [25] such as front-end and intermediate video, image and graphics processing, and oversampled or modulated audio where in excess of 100 000 scalar samples have to be processed in a similar time frame. High throughput algorithms typically execute one or several algorithmic kernels on a massive set of data. Hence they are characterized by a deeply nested loop structure of which the inner loops or repetitive kernels are executing a restricted set of operations iteratively on the large set of pixels. This leads to a distinction between highly multiplexed data path (HMDP) processors executing low throughput algorithms and lowly multiplexed data path (LMDP) processors executing high throughput algorithms.

2) *HMDP Processors*: In CMOS technology, low throughput algorithms have typically of the order of  $20 \cdots 500\,000$  clock cycles available for executing the concatenation of irregular flow graphs whereby the number of operations per cycle is typically lower than ten. All

of this naturally leads to architectures consisting of a few concurrently operating HMDP operators with a rich instruction set, controlled by a single thread sequencer with instruction and status pipelines to keep the clock rate high enough. In view of the low data bandwidth, arrays can be stored in a centralized memory architecture while register files in the datapath store intermediate results.

In this class of HMDP processors we can distinguish three types.

- 1) A commercial, programmable DSP processor which, in most high volume products, is a Harvard architecture consisting of a fixed-point core, a separate program memory and data memory and fixed I/O peripherals. The core has an encoded instruction set tuned to the arithmetic sum-of-product operations and regular memory accesses typical for convolution type DSP algorithms. The processors are characterized by a very heterogeneous register architecture. Actual C compilers, oriented to 32-bit RISC architectures with central register file, produce very poor machine code [26] for DSP processors. Hence DSP designers program them in assembly which leads to a serious design bottleneck and a strong need for rethinking C compilers for DSP processors.
- 2) At the other end of the spectrum is a microcoded very long instruction word (VLIW) processor with a global single thread hardwired controller and data memory. Such a processor is optimized to execute one particular DSP algorithm. The datapath is a customized network of horizontally programmable execution units (EXU's) such as an ALU, MPY-ACC, ACU, ROM, RAM, ... This allows for a higher degree of parallelism than with processors of category 1. The instruction word is not encoded as each EXU can be controlled independent of the others. Instructions are generated by a multibranch microcoded controller optimized for the specific algorithm to be implemented. Efficient high level synthesis compilers exist to interactively synthesize such processors directly from SDF algorithms and their performance constraints. Examples are Cathedral-2 [27] and MISTRAL-2 [11]. Such processors have been successfully used as components in base band telecommunication systems.
- 3) The disadvantage of processors of category 1 with respect to those of category 2 is the overhead in area, power and performance. A disadvantage of category 2 processors is that they lack programmability. Recently, a trend is noticeable in telecom and consumer products to develop application specific instruction set processors (ASIP's) with specialized instructions for a specific domain (e.g., GSM, DECT, Digital Audio, ...) [28]. An ASIP combines reuse of hardware with performance, small area and low power. Key to the success of this concept is the availability of simple ways to generate efficient C or DFL code for ASIP's based on a description of the

instruction set, the architecture and the application to be implemented. This is an intensive topic of research [28]–[31].

3) *LMDP Processors*: A high-throughput algorithm is characterized by a set of repetitive computation intensive kernels for which only a few clock cycles are available to execute one iteration. This means that the multiplexing degree of the operations is small and, since no cycles can be wasted, decision making must be done within a single clock cycle by inserting local control in the datapaths. Such algorithms are very typical for image and video processing but they occur also in CDMA transceivers after modulation with pseudo-noise code [see Fig. 2(a)] or in digital communication systems where the IF part is now also digitized in order to save costly analog filtering components [32].

Obviously, such algorithms are poor candidates for implementation on highly programmable datapath operators with pipelined controllers in view of the performance, area and power overhead. Rather, they give rise to an architecture consisting of a pipelined network of application specific datapaths tuned to a time folding of the kernels to be executed [33]. A global controller only schedules the sequencing of the operations. Decisions are made inside the datapaths. Such processors, which are often very heavily pipelined and retimed are also called *hardware accelerators* and obviously need to be synthesized directly into hardware.

Moreover, since the required data throughput often exceeds the memory bandwidth, arrays must be stored in distributed memory locations and determining the optimum memory architecture is an essential part of the design process [34]–[36].

Recently design methodologies [37]–[39] and synthesis techniques for LMDP architectures [40], [41] have been published. In addition, several high level synthesis tools for LMPD processor compilers have been reported such as Cathedral-3 [33], PHIDEO [1], and HYPER [42]. Synopsys' Behavioral Compiler is also suited for this kind of applications. The input to such compilers is the SDF code of the different SDF functions to be multiplexed and the timing and clock frequency constraints. In most cases the output is a structural VHDL description of the accelerator. The pager design in Fig. 2(b) includes three Cathedral-3 LMDP processors for the IF processing, matched filtering and despreading. They take up quite some silicon area as is visible from Fig. 2(c), which shows the layout of the pager.

Notice that the trend to low power, low voltage systems also leads to such architectures since low power requires less multiplexing and distributed memory architectures. It should indeed be noted that the data fetching from central memory, characteristic for programmability, consumes one order of magnitude more energy than executing a local operation [43] and that multiplexing consumes more power [44] due to loss of signal correlation. Therefore, HYPER [42] combines accelerator synthesis with low power optimization.

**Table 1** Typical Hardware Components of a Telecommunication System and Their Role in the Implementation of the Heterogeneous Specification

Component class	Implements	Compiler	Spec
HMDP	Low data-rate DSP	(Retargetable)	Assembly
DSP Processor	Slow control loops	code generator	C
	Appl. spec. algo's	High level synth.	DFL
LMDP	High data-rate DSP	High level synth.	C, DFL
DSP Processor		RT level synth.	VHDL
Micro controller	User interface	C compiler	Multithread C
	Slow control loops		PSM
Hardware FSM		RT synthesis	
Peripherals	Usually FSDMs:	RT level synth.	VHDL
	- clock generators	Asynchronous	PSM
	- DMA blocks	synth.	
	- ...		
Communication blocks & memory	Internal & external communication	Memory mgmt synth. (A)synchronous	Data-sheets STG
	Storage & buffering	Interface	CSP
	of shared variables	synth.	

Hence the tradeoff between programmability and hardware is not only dictated by data throughput but also by energy considerations.

4) *Control Loops and UI*: Control dominated specifications are of a very different nature as they consist of interacting state machines rather than of a sequence of arithmetic operations to be scheduled within a strict time constraint. They can be mapped on a microcontroller if there is a strong need for future product redefinition or an easy migration into the next product generation. There is a strong trend to follow this road although synthesis of communicating hardware FSM's may be cheaper in area and power. In the latter case, existing RT-level synthesis can be used. In the first case software synthesis techniques such as advocated in [45] can be used, or use must be made of a RTOS.

5) *Conclusion*: Table 1 summarizes the component classes discussed above. As all functions in column 2 are part of a system specification, actual system implementations will contain instances of all these classes. The communication protocol between the processors will have to be implemented using the communication blocks (CB's) possibly storing shared variables in data memory.

From column 4, it follows that the dominant specification language of the telecommunication system designer is C or a dataflow language such as DFL for the main signal path whereas FSM's and PSM's are usually described in VHDL or expanded into it from a PSM description such as SpecCharts [18]. For the description of communication channels and communication protocols other formalisms such as timing diagrams [46], extended signal transition graphs (STG) [47], and CSP [48] must be considered. A CAD environment for telecommunication systems must be able to encapsulate these specification languages and provide a communication paradigm between them. This will be discussed in the sequel of this paper.

### C. The System Design Process

The system design process must bridge the gap between the heterogeneous functional specification, such as in

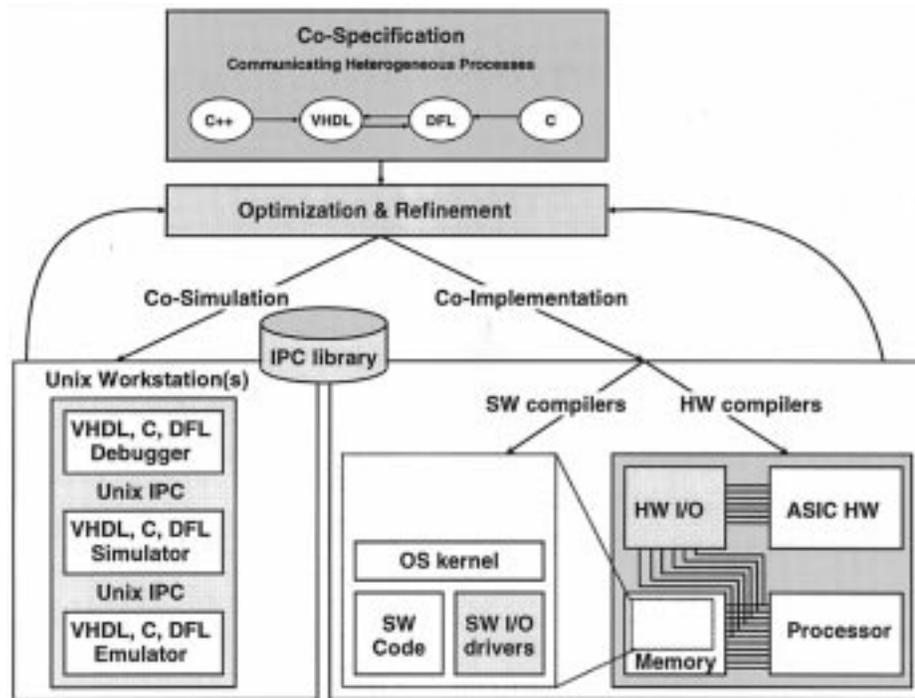


Fig. 3. Architecture of a design environment for co-design of telecommunication systems as under development in IMEC.

Figs. 1(a) and 2(a), and the heterogeneous implementation, such as in Figs. 1(b) and 2(b). Fig. 2 illustrates this for the design of a spread spectrum based pager. Fig. 2(a) is a functional specification similar to the one in Fig. 1(a) and Fig. 2(b) shows a possible implementation of it.

Below we discuss the different activities that must be performed to bridge this gap and give an overview of how they are implemented in the CoWare system design environment, under development at IMEC. Fig. 3 shows the architecture of the CoWare environment.

- 1) The functional specification [Fig. 2(a)] must only describe loose abstract communication between concurrent program threads described in a host language most appropriate for the simulation or later compilation step. This description must be executable in order to validate the system test plan.

In the CoWare design environment the functional specification is based on the concept of communicating CoWare processes. Each primitive CoWare process (symbolized by an ellipse in Fig. 3) encapsulates concurrent program threads in a host language of choice as will be discussed in Section III.

- 2) An important concept of CoWare is that basically no distinction is made between co-simulation and co-implementation. Both are based on the concept of refining the specification for implementation, reusing existing compilers, emulators, and simulation processes. The complete refinement process can be represented using the same data model.
- 3) The first step in the implementation process is to optimally *allocate* processors. This means selecting

the number and types of processors needed to implement the functional specification. The next step is to possibly *merge* functional specifications and to *assign* them to an allocated processor. This merging and assignment step implies the *hardware/software partitioning* step. In the applications we target, this step is hard to automate but relatively easy to perform interactively by designers for the following reasons:

- The number of processors allocated is limited (5–15 components). The types of off-the shelf processors used and documented in a design team is very restricted because of the steep learning curve to insert a new processor as well as the high intellectual property cost involved.
- The merging of signal path functions and the assignment is enforced by flexibility, throughput and power requirements and, as discussed before, is based on coarse grain functional specifications. This limits the choices considerably.
- System design teams seldom design from scratch. Most designs are incremental and a lot of the decision criteria like rough timing estimates are available in the design team which leads rather quickly to a partitioning close to the optimum for the given (restricted) component library available to the team. The key issue here is interactivity [22], [23]. This requires a design environment allowing for quick estimates of timing, area, and power by using sub-optimal modes of the component compilers as estimators whereby a key

ingredient is a quick synthesis and evaluation of the CB's.

In the example in Fig. 2 the high-throughput functions, after merging functions with the same data rate, are assigned to an allocation of three LMDP processors. The slow synchronization loop, and the output data formatting are done on an ARM6 microcontroller core. A clock synthesizer and pseudo-noise code RAM, to store the spreading codes, complete the allocation and assignment.

4) Today's synthesis tools and compilers allow us to synthesize or program the four first processor component classes from Table 1 once the global system architecture has been defined. For mapping concurrent program threads on a single-threaded component, software synthesis techniques or a RTOS must be used. However, Fig. 2(b) clearly shows that the availability of these component compilers is necessary but not sufficient.

5) The most essential step, is to generate the necessary hardware and software to implement the abstract communication paradigms between the functional blocks in the specification. This is illustrated clearly in Fig. 2(b) where the shaded area indicates the communication hardware blocks resulting from the allocation and assignment step.

During the implementation phase we must be able to consistently use the same data model to refine the abstract channels systematically into the communication hardware and software. CAD research in the area of interface synthesis is remarkably underdeveloped. A good outline of the problems and some solutions can be found in [16, ch. 8] and in [49], [47], [50].

In the CoWare design environment the SYMPHONY toolbox provides a methodology for interface synthesis. In this way automatic generation of hardware/hardware interfaces and hardware/software interfaces, including the generation of software drivers in programmable processors is possible. This is an essential part of hardware/software co-design and will be discussed in detail in Section IV.

6) After the compilation of all components, all hardware is available as structural VHDL and all software for the processors is in C which can be compiled on the host compiler of the programmable components, possibly making use of an RTOS. The final step is to link all the synthesis and hardware descriptions to drive commercial back-end tools to generate layout.

In such a system design environment it must be possible to *reuse existing designs* which can be hardware components with a fixed predefined communication protocol, such as programmable processors, or predesigned functional specifications for which a synthesis path for implementation exists. The latter also implies a *design for reuse* strategy in that a strict separation is observed between functional behavior and communication. Design for reuse also im-

plies that it must be possible to reuse existing component compilers and simulators without modifying their internal behavior.

The above methodology has been tested successfully in several designs. In the next section we discuss the basic CoWare data model underlying this design environment and discuss its role in co-design for digital communication systems.

### III. THE COWARE DATA MODEL

From the above requirements a data model and language has been developed on which the CoWare system design environment is built.

Modularity in the specification is provided by means of *processes*. Processes contain *host language encapsulations* which are used to describe the system components. 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 interprocess communication semantics is based on the concept of the *Remote Procedure Call (RPC)*[51, ch. 18]. The data model is hierarchically structured and allows to refine channels, ports, and protocols into lower level objects, adding detail. We refer to the most abstract object as a *primitive* object. An object that contains more implementation detail, is referred to as a *hierarchical* object.

In the next section, we continue with a more detailed discussion of the primitive objects. The hierarchical objects are used to refine the communication behavior of the system and are discussed in Section III-C. Finally, in Section III-E, we explain how the CoWare data model meets the requirements for heterogeneous system design put forth in Section II.

#### A. Once Over Lightly

The simple example in Fig. 4 will be used to illustrate the primitive objects in the data model.

- A *process* is a container for a number of host language encapsulations of a component. 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 (CMF) in the pager example of Fig. 2 was initially described by means of a DFL encapsulation (conceptual level). During refinement, this DFL encapsulation was compiled to a C language encapsulation for simulation purposes. It was then compiled by the Cathedral compiler into a VHDL language encapsulation, which is the processor implementation of the CMF behavior (structural level).

- A *host language encapsulation* describes a component in a specific host language. Currently C, DFL, VHDL, VERILOG, and CoWare are the supported host languages. A CoWare language encapsulation is used to describe the system's structure. In a CoWare

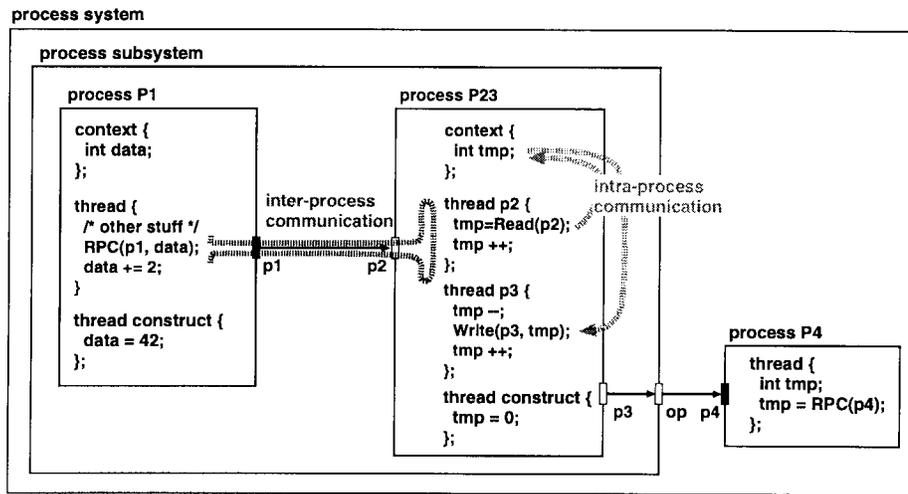


Fig. 4. Illustration of the primitive CoWare data model objects and the two primitive communication mechanisms.

language encapsulation, one can instantiate processes and connect their ports with channels. The other host language encapsulations consist of a *context* and a number of *threads*. The context and thread contain code written in the host language of the encapsulation. 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 interthread (intraprocess) communication. The meaning of threads is explained further on in this section.

In Fig. 4, the processes system and subsystem contain a CoWare language encapsulation. The CoWare language encapsulation of system describes how it is built up from an instance of subsystem and an instance of P4. The processes P1, P23, and P4 each contain a C language encapsulation.

- *Ports* are objects through which processes communicate. A primitive port is characterized by a protocol and a data type parameter. There is one implicit port, the *construct* port, to which an RPC is performed exactly once at system start-up.

In the example of Fig. 4, the process P23 has two primitive ports p2 and p3, next to the implicit construct port.

- *Protocols* define the communication semantics of a port. A primitive protocol is one of *master*, *inmaster*, *outmaster*, *inoutmaster*, *slave*, *inslave*, *outslave*, *inoutslave*. Each primitive protocol indicates another way of data transport. The *in*, *out*, and *inout* prefix indicates the direction of the data. The *master*, *slave* postfix indicates the direction of the control: 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 usually referred to as slave/master ports.

In Fig. 4, master ports are represented by the small shaded rectangles on a process' perimeter. Slave ports are represented by small open rectangles on the

perimeter. The data direction of a port is represented by the arrow that connects to a port. For example, in Fig. 4, port p1 is an outmaster port and port p2 is an inslave port.

A protocol may further have an index set. The indices in the index set are used to convey extra information about the data that is transported. For example the primitive protocol used to model the memory port of a processor will have an index to model the address of the data that is put on the memory port.

- A *thread* is a single flow of control within a process. A thread contains code in the host language of the encapsulation of which the thread is a part. The code in a thread is executed according to the semantics of the host language. 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 (i.e., when an RPC is performed to the slave port). There is one special slave thread which is associated to the implicit *construct* port and can be used to initialize the process.

The process P23 contains two regular slave threads associated to the slave ports p2 and p3, next to the special construct slave thread.

- *Autonomous threads* are not associated to any port and their code is executed, after system initialization, in an infinite loop.

Processes P1 and P4 in Fig. 4 each contain an autonomous thread.

A language encapsulation can contain multiple slave and autonomous threads that, in principle, all execute concurrently.

- A *channel* is a point-to-point connection of a master port and a slave port. Two ports that are connected by a channel can exchange data. Channels can be

uni- or bidirectional. 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 (possibly in-lined) function call. In this way, primitive channels model the basic communication primitives found back in software and hardware description languages.

The example in Fig. 4 contains a primitive channel that connects port p1 of process P1 with port p2 of process P23.

### B. Communication in CoWare

Communication always happens between two threads. Communication between threads that are part of the same process is denoted as *intraprocess* communication. Communication between threads in different processes is denoted as *interprocess* communication.

*Intraprocess (interthread) 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.<sup>2</sup>

In Fig. 4, intraprocess communication occurs, for example, in process P23. The variable `tmp` declared in the context is shared by slave thread p2 and slave thread p3.

*Interprocess (interthread) 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. A master port can be accessed from anywhere in the host language encapsulation (context, autonomous threads, slave threads) with the exception of the construct `thread`.<sup>3</sup> The RPC function returns when the slave thread has completed, i.e., when all the statements in the slave thread's code are executed. In the slave thread (uniquely associated with a slave port), the `Read` and `Write` functions can be used to access the data of the slave port. The `Index` function is used to access the indices of the protocol of the port. The `RWbar`<sup>4</sup> function is used on an `inoutslave` port to determine the actual direction of the data transport. A slave port can only be accessed from within its associated slave thread.

In Fig. 4, interprocess communication occurs, for example, between processes P1 and P23 over the channel that connects ports p1 and p2. When the RPC statement in the autonomous thread of process P1 is reached, the value of

<sup>2</sup>In VHDL, for example, this can be done by making use of synchronization signals. C, for example, does not provide such synchronization mechanisms. However, it also does not support concurrent threads. To actually implement these concurrent threads, one must use a multithread library (see Section IV and Section V) and these libraries then also have methods, such as semaphores, to protect critical sections.

<sup>3</sup>The construct `thread` is activated during system initialization at which point a remote process is not yet guaranteed to be active (and ready to service the RPC).

<sup>4</sup>`RWbar` is a mnemonic for  $R\bar{W}$  as seen from the master side. If the master uses a bidirectional port as input, `RWbar` applied to the connected slave port returns 1.

the local variable `data` is put on the channel, and the control is transferred to the slave thread p2 in process P23. The autonomous thread in process P1 is halted, until the last statement of the slave thread is executed.

### C. Communication Refinement

By using primitive channels, ports, and protocols, the designer concentrates on the functionality of the system while abstracting from terminals, signals, and handshakes. The example of Fig. 4 is modeled in such a way. Once the designer is convinced that the processes of the system are functionally correct, the communication behavior of the system can be refined.

Communication refinement in CoWare is carried out by making the objects involved in the communication (channels, ports, and protocols) hierarchical.

- *Hierarchical channels* are processes that assign a given communication behavior to a primitive channel. The behavioral interface of a hierarchical channel is fixed by the ports connected by the primitive channel. Making a channel *hierarchical*, can drastically change the communication behavior of two connected processes. It can, for example, parallelize (pipeline) the processes by adding buffers. The one property that is preserved by making a channel hierarchical is the direction of the data transport.

In the example of Fig. 5, the primitive channel between processes P1 and P23 is refined into a hierarchical channel with FIFO behavior. The FIFO process decouples the autonomous thread of process P1 and the slave thread p2 of process P2. The effect is that the rate at which process P1 can issue RPC's is no longer determined by the rate at which process P2 can service the RPC's. The FIFO process takes care of the necessary buffering of data.

- *Hierarchical ports* are processes that assign a given communication behavior to a primitive port. The hierarchical port process has one primitive port, which we call the *return* port, that is connected to the primitive port that is made hierarchical. Making a primitive port hierarchical, preserves the data direction (in/out) but may modify the control direction (master/slave). A hierarchical port may convert a master port into a slave port and vice versa.<sup>5</sup>

In the example of Fig. 5, we want to impose a certain data formatting for the data transported over the channel between process P1 and the FIFO process. This is achieved by making the primitive ports, p1 and left, hierarchical. The `format` process that refines port p1 might, for example, add a cyclic redundancy check (CRC) to the data that is transported. The `unformat` process that refines port left of the FIFO process then uses this CRC to determine whether the received data is valid. The actual data and the CRC are sent sequentially over the same primitive channel. As a

<sup>5</sup>If it does, both ports of the channel have to be made hierarchical in the same refinement step to preserve consistency of master to slave connections.

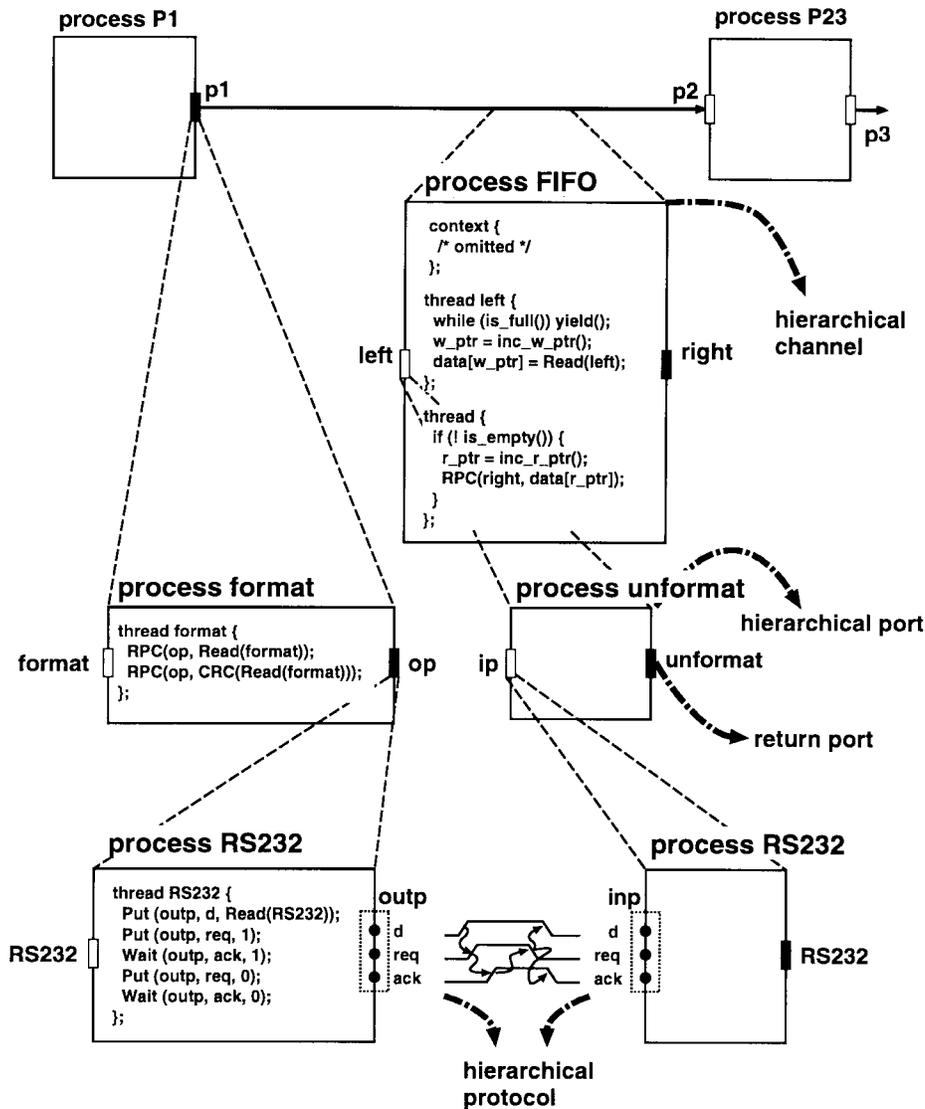


Fig. 5. Communication refinement in CoWare is performed by introducing hierarchical channels, ports, and protocols.

consequence, the data rate between the format and unformat process is twice the one of process P1.

- *Hierarchical protocols* refine primitive protocols with a timing diagram and the associated I/O terminals. Primitive protocols provide a classification of all hierarchical protocols. A primitive protocol determines the communication semantics, but not the communication implementation: it does not fix the timing diagram used in the communication. Hierarchical protocols refine primitive protocols with a timing diagram and the associated I/O terminals. Hierarchical protocols are high level models for alternative implementations of a primitive protocol: they preserve both data direction and control direction of the primitive protocol.

To access the terminals of the hierarchical protocol, a hierarchical port is introduced at the same time. The terminals can be accessed from within the thread code by using the functions Put, Sample, and Wait. In the example of Fig. 5, the primitive protocol of the op port and ip port of the format and unformat process

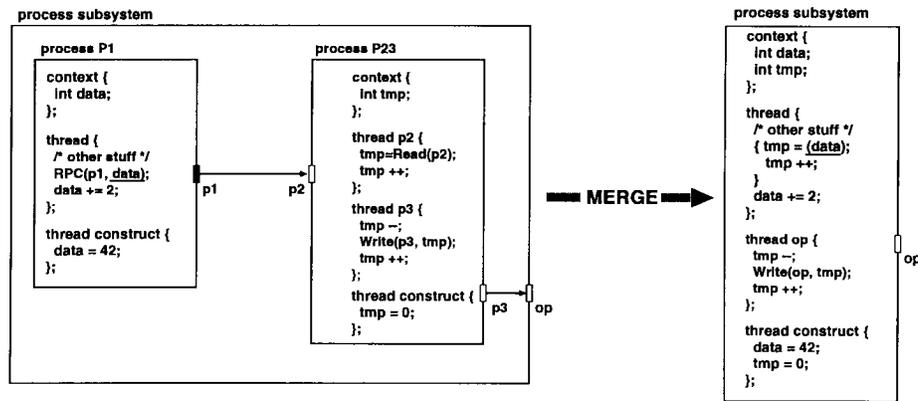
are refined into an RS232 protocol. In the RS232 hierarchical port, an RPC issued in the format process on the op port is converted into manipulations of the terminals according to a timing diagram.

#### D. Communication in CoWare Revisited

Hierarchical channels and ports, being processes, can be removed from a system description by expansion (or flattening). The result is a CoWare description that contains only process instances of which the primitive ports (possibly with hierarchical protocols) are connected by primitive channels. In such an expanded CoWare system description, the three basic communication mechanisms summarized in Table 2 can occur. The first two were already introduced in Section III-A. Interprocess communication with a hierarchical protocol was introduced in this section.

#### E. Justification of the CoWare Data Model

- 1) *Merging of Processes*: Due to the selection of RPC as interprocess communication, the classification of protocols



**Fig. 6.** Merging process instances of the subsystem process, results in a new host language encapsulation for the subsystem process.

**Table 2** Communication Mechanisms Supported by CoWare

Intra-process communication	Inter-process communication	
	primitive protocol	hierarchical protocol
Context	Remote Procedure Call	Terminals, timing diagram
Shared variables	Master port: RPC() Slave port: Read(), Write() Index(), RWbar()	Input: Sample(), Wait() Output: Put()

and the structuring of a process in encapsulations with context and threads, a process merge transformation can be implemented. The goal of this transformation is to combine a number of process instances, that are described in the same host language, into a single host language encapsulation that can then be mapped by a host language compiler onto a single processor. In the process of merging, all remote procedure calls are in-lined: each slave thread is in-lined in the code of the thread that calls it through an RPC statement. Because of the semantics of RPC communication, this transformation does not alter the behavior of the original system, provided that care is taken to avoid name clashes. The result of merging is a host language encapsulation that:

- contains a single context—the contexts of the different processes are merged into one;
- contains a single construct thread—the construct threads of the merged processes are concatenated;
- contains all the autonomous threads from the merged processes;
- possibly contains slave threads through which RPC requests from external processes, that are not involved in the merger, are serviced;
- possibly contains RPC calls to slave threads in external processes, that are not involved in the merger.

Fig. 6 shows the effect of merging the process instances in the subsystem process of the example in Fig. 4. The subsystem process on the left-hand side has a CoWare language encapsulation. After merging the instances in that encapsulation, we obtain the C language encapsulation, shown at the right-hand side of Fig. 6, which is added to the subsystem process.

The benefit of merging processes is that the in-lining transformation eliminates the overhead that accompanies execution of (remote) procedure calls. It further reduces the number of concurrent threads and, therefore, the overhead that accompanies the switching between threads. Finally, it allows the host language compilers to optimize over the boundaries of the original processes.

2) *Design for Reuse and Reuse of Designs:* The port and protocol hierarchy provides a clear separation between functional and communication behavior. Traditionally, the description of a component contains both functional and communication behavior in an interleaved way. When such a component has to be reused, in an environment other than it was intended for, the designer has to change those parts of the description of the component that have to do with communication. In CoWare, a component's behavior is described by a process that makes use of RPC to communicate with the outside world. Such processes can be connected with each other without modifying their description (modularity). By using primitive ports and primitive protocols, the designer concentrates on the functionality of the system while abstracting from terminals, signals, and handshakes. 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 involved. This fixes the timing diagram and terminals used to communicate over that port. 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. Again this is achieved without modifying the description of any of the processes involved. Because of this property it is feasible to construct libraries of functional building blocks and libraries of CB's that are reusable: they can be plugged together without modifying their description. After blocks have been plugged together, any communication overhead (chains of remote procedure calls) can be removed by inlining the slave threads that serve the RPC's. The result is a description of the component in which function and communication are interleaved seamlessly and which can be compiled into software or hardware

as efficiently as a description in the traditional design process.

The above 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 designs”) because the timing diagrams according to which a processor communicates are abstracted in it.

Hence, the choice for RPC as basic communication mechanism is motivated by the following observations.

- *Abstract.* By using RPC, a designer concentrates on functionality without bothering about signals and handshakes. The signals and handshakes can be introduced later on in the design process by refining the primitive protocol in a hierarchical protocol.
- *Minimal.* Other communication mechanisms can be modeled by making use of RPC.
- *Modular.* Functional and communication behavior of a process are separated.
- *Removable.* Due to the selection of RPC as interprocess communication, the classification of protocols and the structuring of a process in encapsulations with context and threads, the merge transformation (Section III-E1) can be implemented to remove modularity.

The above observations can be summarized in the following statement: *RPC supports design for reuse and reuse of designs.*

#### IV. IMPLEMENTATION

As discussed in Section II-C, the input to the implementation refinement process is a functional specification: a CoWare encapsulation consisting of a number of process instances (i.e., host language encapsulations), exhibiting both intraprocess and interprocess communication behavior. In a first step, *allocation* is performed. In this step the number and type of processors are selected that will serve as the target for implementing the input specification. After allocating the necessary processor resources, an *assignment* step is performed. In this step each process instance of the input specification is assigned to one of the allocated processors. Only process instances with the same host language encapsulation can be assigned to the same processor.<sup>6</sup>

The remainder of the implementation path consists of the following steps:

- 1) All process instances bound to a single processor are merged (see Section III-E1). This results in a system with a one-to-one mapping between (merged) processes and allocated processors.
- 2) The host language encapsulation from each (merged) process instance now has to be compiled with the appropriate host language compiler onto its processor

<sup>6</sup>Host language transformations (either by the designer or tool-supported) may be required to realize a particular assignment.

target. Since we reuse existing compilers, the CoWare concepts of autonomous threads, slave threads, and shared context are not always directly supported. In that case one has to decide how these concepts can be implemented using the selected host language compiler.<sup>7</sup>

- 3) The result of each host language compiler is encapsulated so that it can be connected to the rest of the system. This involves the implementation of the interprocess communication and making the (generated) processor protocol compatible with the processors to which it is connected.

The complexity of this step can vary widely depending on the processor and host language compiler used.

- a) The host language compiler can generate processors with a user-defined interface (protocol), e.g., the SYNOPSIS design compiler. In this case, the interface can be made compatible with the other processors before entering the host language compiler. This is achieved by making the ports/protocols of the (merged) host language encapsulation hierarchical. These hierarchical ports/protocols are then merged with the rest of the host language encapsulation and compiled.
- b) The host language compiler generates a processor with a fixed given interface, e.g., the Cathedral DSP compiler. In this case the ports/protocols of the generated processor have to be made hierarchical to be compatible with the other processors.
- c) An off-the-shelf processor is used. This case is similar to the case above.

For the protocol conversion between two (hardware) processors an automated path is proposed in the SYMPHONY toolbox. This automated protocol conversion is implemented by standardizing on an well-chosen intermediate protocol: the *synchronous wait protocol*. However, according to the CoWare philosophy the designer can always discard this proposal and use a different protocol conversion strategy. The synchronous wait protocol and the protocol conversion based on it will be explained in Section IV-A.

The generation of the hardware/software interface to implement communication between a C language encapsulation compiled on a programmable processor and a VHDL language encapsulation compiled in to hardware, is also automated in the toolbox SYMPHONY and will be discussed in Section IV-B.

The implementation of multiple (concurrent) threads and the interthread (intraprocess) communication will be discussed in Section IV-A, for the hardware case, and in Section IV-C, for the software case.

<sup>7</sup>Alternatively, one might restrict a host language encapsulation so that it can be directly compiled.

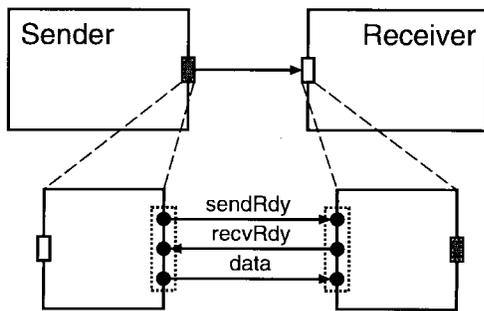


Fig. 7. Communication implemented using a synchronous wait protocol.

#### A. HW/HW Communication

The intermediate protocol used for communication between two hardware processors is defined in Section IV-A1 for the case where the communication partners operate under the same clock. In Section IV-A2 we evaluate the performance of the protocol scheme. In Section IV-A3, we describe how synchronization is handled for components that operate on different derived clock frequencies, and on unrelated clocks. In Section IV-A4 we explain how RPC calls can be implemented using this protocol scheme. Finally, in Section IV-A5 we show that this protocol scheme can also be used for communicating between different threads of a single VHDL host-language encapsulation.

1) *Synchronous Wait Protocol*: To define a common protocol scheme, we start from the observation that in many designs today the communicating components are synchronous and the clocking discipline used in the system is a set of derived clocks that have an exact phase relationship with each other. We further assume that the designs are positive edge-triggered and the components have registered outputs. The latter assumption is in fact common with today's commercial hardware synthesis tools [52]. Based on these assumptions, a communication channel can be implemented in hardware using a simple synchronous transfer protocol called a *synchronous wait protocol*. In the remainder of this section we assume that both partners operate on the same clock.

In the synchronous wait protocol, the sender and receiver partners synchronize the communication by a pair of `sendRdy` and `recvRdy` signals, as shown in Fig. 7. The communication follows the automaton depicted in Fig. 8, showing all possible signal transition sequences that are compliant with the synchronous wait protocol. Suppose the initial state of the communication is S1. This means that initially neither the sender nor the receiver are ready to set up a data transfer, as indicated by the signals `sendRdy` and `recvRdy` being low. If at a certain moment the sender becomes ready first, the sender partner sets its `sendRdy` signal high, denoting it has put valid data on the `data` lines. The sender will then enter into the "wait state" S3 until the receiver is ready. Notice that in this wait state, the `data` lines as well as the `sendRdy` signal are not allowed to change: one can only leave state S3 when the `recvRdy` signal goes high. When the receiver becomes

ready, as indicated by the signal `recvRdy` being high, the communication arrives in state S4 and the transfer is assumed to be completed in that clock cycle; thus the completion of the data transfer is left implicit. Similarly, if initially the receiver becomes ready first, the receiver enters into the wait state S2 until the sender is ready. In this wait state the receiver has to keep the signal `recvRdy` invariantly high. Finally, when both partners are ready, the communication arrives in state S4 and a data transfer takes place. A third possibility is that initially both partners become ready at the same time. The communication then directly jumps from state S1 to state S4. Notice that once arrived in state S4 and both partners remain ready, a data transfer can take place every new clock cycle.

2) *Performance Evaluation*: Despite its simplicity, the synchronous wait protocol offers several important advantages. One advantage is that the completion of communication is implicit. This means that when both partners are "ready," the communication behaves like a RT operation. "Burst" transfer modes, where the sender transfers consecutively a sequence of data to the receiver, can be implemented very efficiently. In fact, new data can be transferred at a sustained rate of one item per cycle. This is in contrast to, i.e., a handshaking protocol like the four-phase request-acknowledge scheme where the completion of the transfer has to be explicitly acknowledged. The four-phase handshaking protocol is widely used in asynchronous circuit implementations [53], [54] synchronize communication in the absence of a clock. But when employed in a synchronous positive edge-triggered design, this request-acknowledge scheme requires at least two clock cycles for each transfer to account for the explicit acknowledgment. This problem can be partly circumvented if the channel control logic is clocked at twice the clock frequency by using both phases of the clock to trigger the logic. However, this will make the circuit more difficult to test and less amendable to conventional resynthesis by synchronous hardware synthesis techniques.

3) *Different Clocking Disciplines*: When the components are clocked by different clock frequencies, a small *channel adapter* handles the clock conversion (see Fig. 9). When the communicating components operate under clocks that are derived from the same global system clock and clock skew can be neglected, this channel adapter is in effect a frequency converter, and it can be implemented using simple state machine logic. When the communicating components operate under unrelated clocks or when clock skew cannot be neglected, the channel adapter is implemented using an internal handshake protocol and synchronizers [55].

Using this channel adapter approach, the designer can integrate an already synthesized (hardware-level) component from a previous design application into a new custom embedded system architecture. The main requirement is that the reused component has been designed in compliance with the synchronous wait protocol. If this is not the case, a wrapper can be placed around the component to convert the internal protocol to the synchronous wait protocol, using for example similar techniques as described in [53].

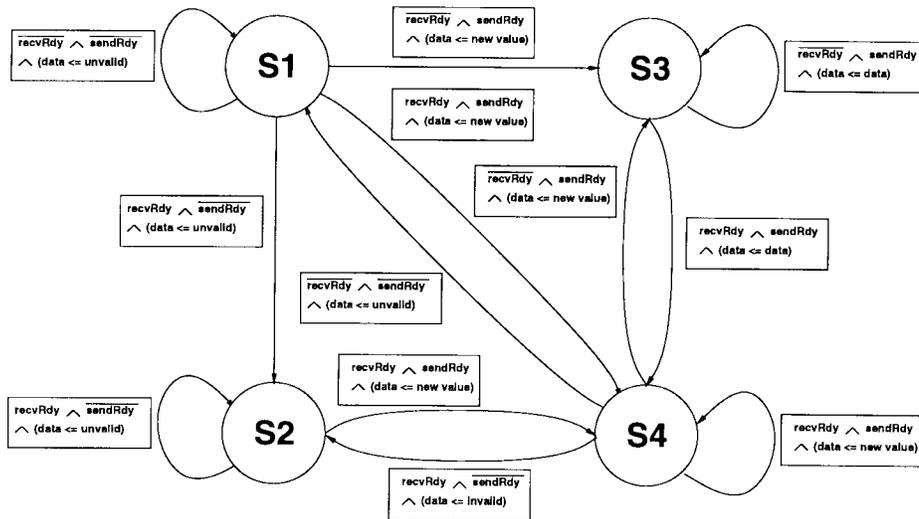


Fig. 8. Automaton of the synchronous wait protocol.

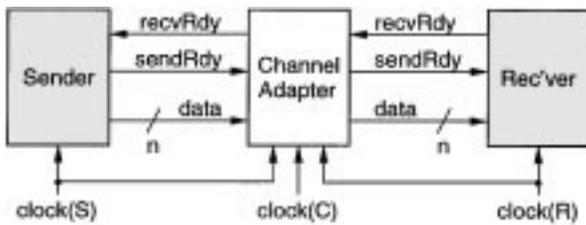


Fig. 9. Channel adapter for frequency conversion.

#### 4) Implementation of RPC Interprocess Communications:

In the case of an RPC communication channel we classify the partner that controls an outmaster, outslave, or master port as the sender partner, and the partner that controls an inmaster, inslave, or slave port as the receiver partner of that particular channel [see Fig. 10], where the case is shown of a channel connecting an outmaster port with an inslave port. The outmaster partner implements an RPC operation by setting its  $\text{sendRdy}$  signal high and placing valid data on the  $\text{data}$  lines. The inslave partner then exits its “wait state” and begins executing the corresponding slave thread. During this time, the inslave partner sets its  $\text{recvRdy}$  low, keeping the outmaster partner into a wait state. This ensures synchronization. If the inslave partner is done executing the slave thread, as indicated by the  $\text{recvRdy}$  signal being high, then the RPC is assumed to be completed in that clock cycle.

5) *Intraprocess Communications:* In the case of mapping a host-language encapsulation described in VHDL onto an application specific hardware processor, SYMPHONY supports the designer with a VHDL package for issuing intraprocess communication using the synchronous wait protocol. This package provides a generic channel type and  $\text{send}$  and  $\text{receive}$  procedures for specifying rendezvous communication on objects of that particular channel type. The designer can then declare a number of channel type objects in the context of the VHDL host-language encapsulation, and use the  $\text{send}$  and  $\text{receive}$  procedures to synchronize and communicate data between the dif-

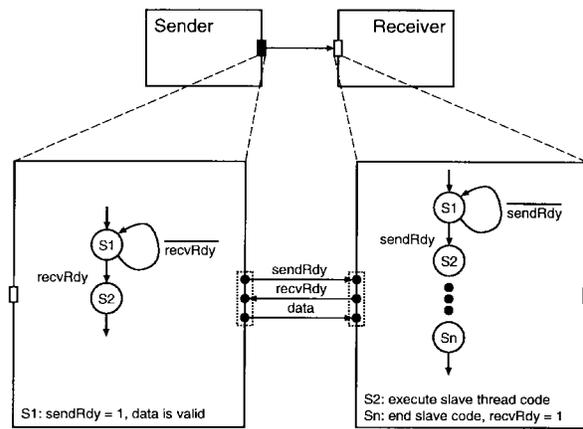


Fig. 10. outmaster-inslave channel transfer.

ferent threads of the VHDL host-language encapsulation. The channel type objects are implemented according to the synchronous wait protocol described above, and the  $\text{send}$  and  $\text{receive}$  procedures are implemented in the same way as an RPC on a outmaster port and inmaster port, respectively. The  $\text{send}$  procedure is implemented by setting the  $\text{sendRdy}$  signal high and placing the argument data on the  $\text{data}$  lines [see Fig. 11(a)]. If the receiver is not yet ready, as indicated by the input  $\text{recvRdy}$  signal being low, then the  $\text{send}$  procedure implements a “wait state” until the receiver is ready. This ensures synchronization. When the receiver is ready, as indicated by the input  $\text{recvRdy}$  signal being high, then the  $\text{send}$  procedure is assumed to be completed in that clock cycle. Similarly, a  $\text{receive}$  procedure is implemented by setting the  $\text{recvRdy}$  signal high [see Fig. 11(b)]. If the sender is not yet ready, as indicated by the input  $\text{sendRdy}$  signal being low, then the  $\text{receive}$  procedure implements a wait state until the sender is ready. When the sender is ready, as indicated by the input  $\text{sendRdy}$  signal being high, then the  $\text{receive}$  procedure latches and returns the data.

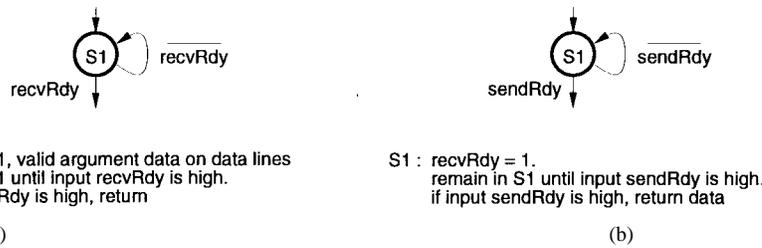


Fig. 11. State machine representation of (a) send procedure and (b) receive procedure.

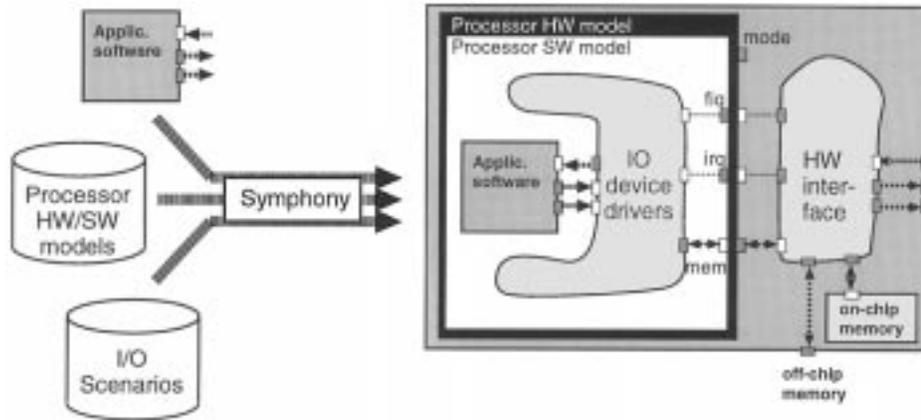


Fig. 12. SYMPHONY uses a library of processor models, and I/O scenario models to generate a hardware/software interface for a given application process.

The final implementation of the VHDL encapsulation will then consist of a number of FSMD's (one FSMD per thread) communicating with each other via channels using CSP rendezvous communication actions. Due to the synchronous wait protocol, synchronization between the different FSMD's—that may be clocked differently—is then automatically handled.

### B. SW/HW Communication

To implement the RPC calls between a software host-language encapsulation and a hardware host-language encapsulation, we replace the software host-language encapsulation by a new hardware host-language encapsulation that is behaviorally equivalent. Hence the problem is shifted to implementing the same interprocess communication in hardware (see Section IV-A). This is shown in Fig. 12. To facilitate this, the newly created hardware host-language encapsulation builds an architecture template around the processor core implementing the RPC calling mechanism across the software/hardware boundary. There are three main components to this architecture template: the processor core itself, a memory structure for storing the program instructions and run-time data, and a hardware I/O unit that implements the RPC communication interface to the external environment. To customize the architecture template SYMPHONY reads in the software host-language encapsulation itself, a *software model* and *hardware model* of the processor core and a library of *I/O scenario models*.

A *hardware model* of the programmable processor core consists of a VHDL host-language encapsulation<sup>8</sup> that formalizes the information that is available in the hardware section of the data sheet of the programmable processor core. In Fig. 12 this hardware model is represented by the rectangle marked "Processor HW model" and the ports at the outside of the rectangle's perimeter. The VHDL host-language encapsulation of the hardware model is characterized by a behavioral interface that is conform with the hardware interface of the programmable processor. All ports have hierarchical protocols: they consist of terminals and a timing diagram (including timing constraints). The hardware model may also contain a VHDL description (either a black box, a simulation model, or the full description of the processor core).

A *software model* of the programmable processor core consists of a C host-language encapsulation that formalizes the information that is available in the software section of the data sheet of the programmable processor. In Fig. 12 this software model is represented by the rectangle marked "Processor SW model" and the ports at the inside of the rectangle's perimeter. The C host-language encapsulation of the software model is characterized by a behavioral interface that is conform with the software interface of the programmable processor core. All ports have primitive

<sup>8</sup>When we use an instruction-set simulator for simulation purposes, the hardware model of the core consists of a C host-language encapsulation, in which the API of the instruction-set simulator is used to communicate with the core.

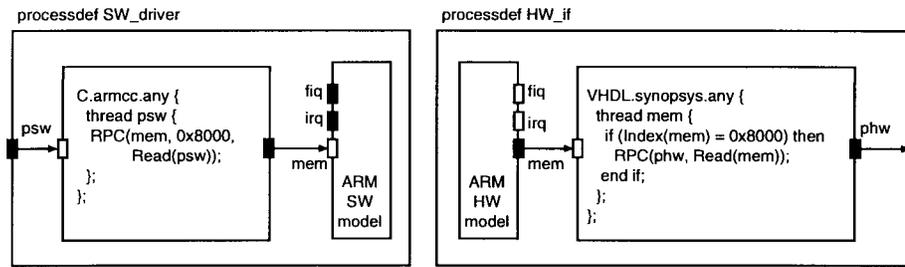


Fig. 13. Example of a memory mapped I/O scenario.

protocols. The software model identifies, for example, what ports can be used to get data in or out of the processor core (memory mapped, co-processor port, ...), what ports can be used as interrupt ports and what their characteristics are (interrupt priority, maskable interrupt, ...). In addition the software model contains a behavioral description that allows to compile a C process into machine code. For example: functions to manage processor specific actions such as installing an interrupt vector, enabling/disabling interrupts, etc.

An *I/O scenario model* describes one way of using the ports of a specific processor core to map a particular port of the application software to an equivalent port in hardware, thereby crossing the processor core boundary while maintaining the communication semantics. It consists of a software host-language encapsulation and a hardware host-language encapsulation that describe a software I/O driver and the hardware counterpart respectively. An I/O scenario is also tagged with some performance figures that will allow the designer or SYMPHONY to make a decision about what I/O scenario to use for which port. SYMPHONY supports many I/O scenarios:

- *Memory-mapped I/O* provides a data transfer mechanism that is convenient because it does not require the use of special processor instructions, and can implement practically as many input *or* output ports as desired. In memory-mapped I/O, portions of the address space are assigned to I/O ports. Reads and writes to those addresses are interpreted as commands to the I/O ports. “Sending” to a memory-mapped location involves effectively executing a “Store” instruction on a pseudo-memory location connected to an output port, and “Receiving” from a memory-mapped location involves effectively executing a “Load” instruction on a pseudo-memory location connected to an input port. When these memory operations are executed on the portions of address space assigned to memory-mapped I/O, the memory system ignores the operation. The I/O unit, however, sees the operation and performs the corresponding operation to the connected I/O ports.

In Fig. 13, an I/O scenario is depicted that shows how an *outmaster* port *psw* of the application software can be “mapped” to an *outmaster* port *phw* in hardware, thereby using the memory port of an ARM-6 RISC core. The software process encapsulation *SW\_driver* represents the software I/O driver

and copies data from port *psw* to a specific memory address *0x8000*. The hardware process encapsulation *HW\_if* represents the hardware counterpart and checks whether the memory address bus of the ARM equals *0x8000*. If this is the case, data that is residing on the memory data bus of the ARM is copied to port *phw* via an RPC call. For custom embedded architecture synthesis, the number of memory locations assigned for memory-mapped I/O will depend on the number of ports that a software processor component has to “physically” implement. SYMPHONY proposes an assignment of address locations to channels that will result in simple address decoding logic. However, the user can always override the proposed assignment.

- *Instruction-Programmed I/O*. Some processors also provide special instructions for accessing special I/O ports provided with the processor itself. Using this scheme, these special communication ports of the processor are connected to the external channels via the I/O unit.
- *Interrupt Control*. In addition to providing hardware support for memory-mapped and instruction-programmed I/O, the I/O unit also provides support for hardware interrupt control. Interrupts are used for different purposes, including the coordination of interrupt-driven I/O transfers. Different processors provide different degree of hardware interrupt support. Some processors provide direct access to a number of dedicated interrupt signals. If more interrupt “channels” are required, as for example required to support a number of interrupt-driven communication channels, scenarios that make use of interrupt vectors can be used. *Interrupt vectors* are pointers or addresses that tell the processor core where to jump to for the interrupt service routine. In effect, this is a kind of memory-mapped interrupt handling.

Once an I/O scenario model is selected for every port of the software host-language encapsulation, SYMPHONY generates the necessary communication software and the corresponding hardware I/O unit by combining all selected I/O scenarios for the ports. The generated communication software, the software model of the processor core and the software host-language encapsulation itself are merged and compiled with the processor specific C-compiler. The resulting executable will be loaded into the memory structure of the architecture template.

The actual selection of the I/O scenario models for the ports can be done by the designer in an interactive way or it can be done automatically by SYMPHONY. The automatic selection is based on a simple greedy algorithm [56] that looks at the cost associated with each scenario. The algorithm minimizes the overall cost of the hardware/software interface. For example, instruction-programmed I/O will be used first if it is less expensive than memory-mapped I/O. If communication via special programmed I/O instructions is more expensive, or not available, then only memory-mapped I/O will be used.

### C. SW/SW Communication

The problem of implementing interprocess communication between two software process encapsulations assigned to different processors, can be reduced to the problem described in Section IV-B. The two software process encapsulations will be replaced by two behaviorally equivalent hardware host-language encapsulations, in turn reducing the problem to the problem described in Section IV-A.

Intraprocess communication can be realized in software by using shared variables that are declared in the context of the process. Correctness is then the user's responsibility. With this approach, for two slave threads or a slave thread and an autonomous thread to communicate, there is no real problem, as in practice slave threads will be mapped to the interrupt routines of the processor core under consideration. The result of this is that the execution of these two threads will not be interleaved at any time, avoiding the traditional concurrency problems with respect to shared memory communication.

To handle the timely execution, communication and synchronization of two autonomous threads, a (real-time) kernel will be allocated on the same processor core. In recent years, a number of lightweight real-time kernels dedicated to embedded applications have emerged on the market [57]–[60] that are fairly small in size and tuned for performance (e.g., fast context switch and interrupt service). They provide many high-level services to support the runtime coordination of concurrent threads. Services include task scheduling and thread control, and communication mechanisms like semaphores, mailboxes, and queues. Some kernel vendors [59], [60] aim to support a broad portfolio of processors (both DSP's and microcontrollers) by using a kernel architecture that isolates processor specific aspects so that minimal porting is required to support different processors.

However, the currently available real-time kernels are not without limitations. They typically assume a fixed preimplemented multiprocessor architectures on printed circuit boards (e.g., VME-based boards). They cannot directly be used for an application-specific architecture without manual porting of low-level services such as interprocessor communication, and they do not at all support mixed hardware/software solutions since functionalities for communicating and interacting with dedicated hardware components are entirely lacking. As a consequence, real-time kernels are rarely used for embedded system-on-

silicon applications, despite the important functionalities that they provide. Instead, designers currently have to struggle with implementing manually low-level assembly routines to implement communication and task coordination functionalities that are often already a part of a real-time kernel. The implementation of these routines is a laborious and highly error prone task, without a clear methodology to structure the implementation process.

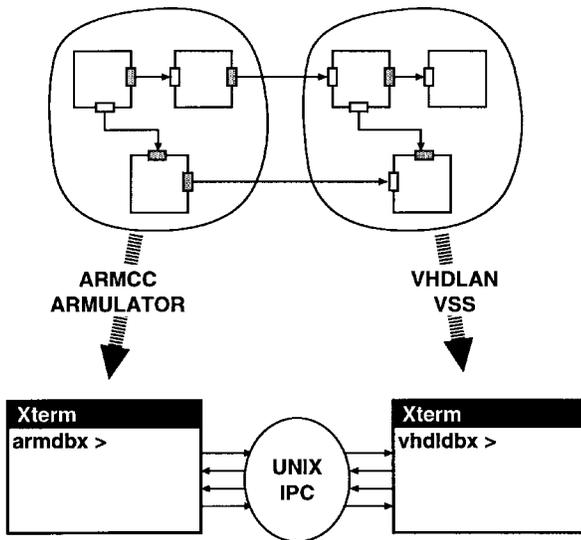
In our earlier work [61], we presented a methodology for providing real-time kernel support for application-specific hardware/software embedded architectures. A key concept in this approach is the generation of a hardware/software communication abstraction layer through the standard architecture and communication protocol. SYMPHONY uses this work by providing a channel communication mechanism for issuing rendezvous communication between threads.

## V. SIMULATION

In the CoWare system design environment, system simulation is considered as an implementation of the system on a work station. The compilation path to implement a simulation, closely resembles the implementation path to silicon. In the latter case, all processes in the system description are ultimately implemented in hardware and the communication between processes is implemented by wires. In the case of simulation the common layer between the process implementations is not hardware but the operating system of the work station. The processes in the system description are implemented by software that runs in a UNIX process on the work station. The communication between the system processes is implemented by UNIX interprocess communication. Fig. 14 shows a typical simulation of a CoWare system. The original system description is shown at the top. The bottom part of the figure shows what the simulation looks like on the screen of a work station. One UNIX process runs the instruction-set simulator (`armdbx`) for the processes assigned to the ARM processor core. The other UNIX process runs the VHDL debugger (`vhdl1dbx`). The communication between these two UNIX processes is implemented over UNIX IPC.

### A. Building a Simulation

The steps required to build a simulation are similar to those for building a silicon implementation. Allocation and assignment in the normal implementation trajectory corresponds to the selection of the appropriate simulators on which to run the system's processes. Examples of simulators are: Synopsys' VSS simulator for VHDL encapsulations of processes, ARMulator for ARM C encapsulations, UNIX processes for high level C encapsulations, etc. After assignment, all process instances (host language encapsulations) assigned to the same simulation-processor are merged (Section III-E1). The resulting merged host language encapsulations now have to be implemented on the selected simulation processors and the communication between the simulation processors has to be implemented.



**Fig. 14.** A typical simulation in the CoWare environment. The CoWare system description at the top is implemented as a set of UNIX processes, each running a native debugger, communicating over the common UNIX operating system layer.

Since all simulators actually run inside a UNIX process, the latter aspect involves two steps [28].

- 1) Making communication links from the simulator to the UNIX process. This is done by making use of the application programmer's interface of the simulator which is available in most state of the art commercial simulators (e.g., in SYNOPSIS' VSS simulator this feature is denoted CLI: C-Language Interface, while in CADENCE's Leapfrog simulator it is called FMI: Foreign Model Import).
- 2) Making communication links from one UNIX process to another through the UNIX operating system. The UNIX OS provides a number of interprocess communication mechanisms which can be used for this purpose: files, pipes, sockets, message queues, shared memory, etc. In terms of CPU overhead, shared memory with semaphores exhibits the best performance [51, Chapter 17]. For this reason the basic UNIX IPC mechanism used in CoWare for co-simulation is shared memory with semaphores. As a consequence, co-simulation in CoWare is restricted to a single machine at the moment.<sup>9</sup>

The implementation of the host language encapsulation on a simulation-processor is mainly complicated when the simulation-processor does not support concurrent threads directly. This is the case when simulating a C language encapsulation in a UNIX process. To solve this problem a multi-task library [62] can be used.

### B. Simulation Requirements

From a user's perspective, the most important factors in simulation are speed and debugging power. Simulation

<sup>9</sup>This is not a restriction of the methodology. By replacing the shared memory hierarchical ports by, for example, TCP/IP ports the simulation can be distributed over a network of several work stations.

speed is required to enable the functional verification of complex system behavior. For example: the functional verification of the tracking and acquisition loop of the spread-spectrum receiver in Fig. 2 requires several tens of seconds real-time simulation which is obviously not realistic with a gate level VHDL simulation of the system. Realistic simulation speeds can be obtained by using more abstract, higher level models of the system (e.g., C or C++ models). However, these have the drawback that one can not observe the detailed behavior of e.g., a processor interface. Therefore current practice is to isolate the problematic part of a system and to simulate that at the detailed level. This requires a lot of effort from the designer's part: manual rewriting the system configuration, construction of an appropriate test bench that activates the problem, etc.

To meet these simulation requirements the CoWare design environment supports mixed abstraction level verification. Selected parts of the system can, e.g., be simulated at a low abstraction level while the rest of the system is simulated at a high abstraction level. The design environment will set up the simulation and take care of the generation of the required intersimulator communication. The advantages of mixed abstraction level simulation are:

- one can zoom-in on problematic behavior by simulating the processes involved on a lower abstraction level;
- processes that are not relevant for the problem at hand, are simulated at the highest abstraction level;
- one can use the original testbench for all simulations.

The abstraction levels at which processes are simulated is determined by the simulation architecture. The CoWare design environment supports the construction of different simulation architectures starting from the original system description.

The speed of the simulation is not only determined by the simulation architecture but also by the level of concurrency in the simulation architecture. This concurrency has to be emulated on a single thread machine, hence the overhead due to switching between concurrent processes. To improve simulation speed, concurrency should be reduced as much as possible. In CoWare this is done by merging processes as much as possible. This reduces task switching by inlining slave threads and reduces UNIX process switching at the cost of intraprocess task switching (which usually is cheaper).

## VI. DESIGN EXAMPLE: A SPREAD-SPECTRUM BASED PAGER

In this section we illustrate the CoWare design methodology on the spread-spectrum based pager system of Fig. 2.

### A. Specification of the Pager

Each block in Fig. 2(a) corresponds to a process implementing a specific function of the pager. This functional decomposition determines the initial partitioning: the finest granularity is determined by the functions in the system.

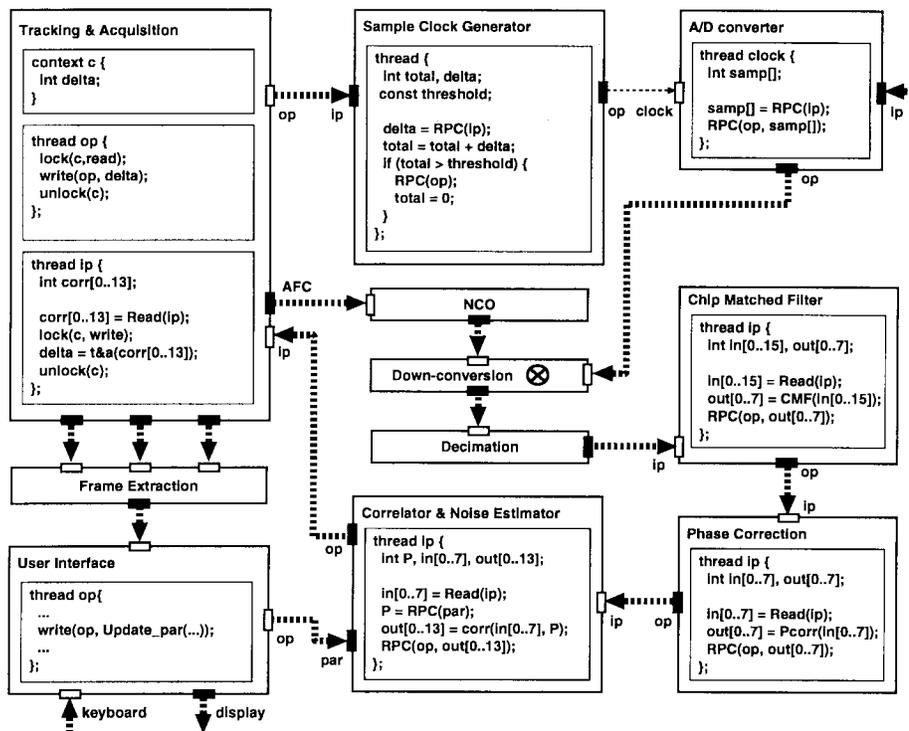


Fig. 15. RPC as a basic communication mechanism between processes.

The arrows in between the processes represent primitive channels with RPC semantics. Fig. 15 shows the RPC communication in detail for part of the pager design. The blocks in the figure correspond to the processes from Fig. 2(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. This thread runs continuously. It performs an RPC over its input port *ip* to the Tracking and Acquisition process to obtain a new value for *delta*. The autonomous thread of the process adds the *delta* parameter to some internal variable until a threshold is exceeded. In this way it implements a sawtooth function. When the sawtooth exceeds the (fixed) threshold an RPC call is issued to the A/D converter process. The autonomous thread of the Sample Clock Generator performs an RPC (gives a sample clock tick) every *threshold/delta* iterations (real clock cycles).

The slave thread *clock* in the A/D converter process samples the analog input, and sends the result to the Down-conversion process via an RPC call. This in turn will activate the Decimation process via an RPC call, etc.

The Correlator and Noise Estimator process contains a slave thread associated with port *ip* to compute the correlation values. This slave thread is activated when the Phase Correction process writes data to the Correlator and Noise Estimator process (i.e., when the Phase Correction process performs an RPC to the *ip* port of the Correlator and Noise Estimator process). The slave thread reads in the data and then performs an RPC to the UI process to obtain a new value for the parameter *par* it requires for computing

the correlation values. Finally, the new correlation results are sent to the Tracking and Acquisition process via an RPC call on its *op* port.

The slave thread in the Tracking and Acquisition process updates the *delta* value for the sawtooth function implemented by the Sample Clock Generator process. It puts the updated value in the context, where it is retrieved by the slave thread *op* which serves RPC requests from the Sample Clock Generator process. In this way the Tracking and Acquisition process influences the frequency of the clock generated by the Sample Clock Generator 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. The locking and unlocking of the context is required to avoid concurrent accesses to the variable *delta*. The *lock*<sup>10</sup> in the slave thread *op* locks the context for read: other threads are still allowed to read from the context, but no other thread may write the context. The *lock* in the slave thread *ip* locks the context for write: no other thread is allowed to write or read the context until it is unlocked again.

Each process is described in the language that is best fit for the characteristics of the function it implements.<sup>11</sup> The dataflow blocks (NCO, Down-conversion, Decimation,

<sup>10</sup>The function *lock* is a high-level primitive provided by the CoWare environment for this purpose (similar to the *send* and *receive* communication primitives discussed in Section IV). The CoWare environment makes sure that implementations for *lock* exist for all implementation targets.

<sup>11</sup>The language used in Fig. 15 is pseudo-code meant to illustrate the RPC concept; it does not correspond to any of the languages mentioned in this document.

CMF, Phase Correction, Correlator and Noise Estimator, and Sample Clock Generator) are described in DFL. The control oriented blocks (Tracking and Acquisition, Frame Extraction, and UI) are described in C.

The sample rate differences between the processes (e.g., Downconversion and Decimation outputs) are modeled by having an appropriate ratio between activation of the slave inputs and RPC's to the master outputs.

### B. Design Process

After the initial specification of the system has been validated by simulation, the designer starts the refinement process.

At this moment it is not yet decided what process will be implemented on what kind of target processor nor is it defined how the RPC communication will be refined. However, the choice of the specification language for each process restricts the choice of the component compiler and in that sense partly determines the target processor. Hence, studying possible alternative assignments of a process to a target processor may require the availability of a description of the process in more than one specification language or a clear guess of the best solution.

1) *Allocation and Assignment*: This step determines what processes will be implemented on what target processor. As explained in Section VI-A, the initial specification shows the finest grain partitioning: a process in the initial specification will never be split over several processors. However, it may be worthwhile to combine a number of processes inside a single processor. This is achieved by merging these processes into a single process that can then be mapped on the selected target processor by a host language compiler. Merging of processes is only allowed when the processes are described in the same specification language. Hence, studying possible alternative mergers may require that for a number of processes (e.g., Correlator and Noise Estimator process) a description is available in more than one specification language. After allocation and assignment, one obtains a description with a one-to-one mapping of merged processes to processors.

In the pager example [Fig. 2(a)] the following allocation, merging and assignment takes place:

- The NCO, Down-conversion, and Decimation processes are merged and mapped in hardware onto an application specific DSP Cathedral-3 [33] processor because the sample rate of the merged processes is identical<sup>12</sup> which implies that they can be clocked at the same frequency. The advantage is that only one clock tree needs to be generated per merged process (i.s.o. one per original process). An additional advantage is that the scan-chains for the processes that are merged can be combined.

<sup>12</sup>The sample rate is  $8 * M$  samples/chip where  $M$  is the oversampling rate. The input rate for the Decimation process is  $8 * M$  complex samples/chip but since it processes the real and imaginary part of each sample concurrently, the Decimation effectively operates at  $8 * M$  samples/chip.

- The CMF and Phase Correction processes are merged and mapped onto a Cathedral-3 processor because their sample rates are identical.<sup>13</sup>
- The Correlator and Noise Estimator process is mapped onto a Cathedral-3 processor. It is not merged with the Phase Correction process because it operates at a four times lower frequency.<sup>14</sup>
- The Sample clock generator is mapped onto a Cathedral-3 processor.
- Tracking and Acquisition, Frame Extraction, and UI are merged and mapped on a programmable processor. For this design an ARM6 processor is chosen. The Hardware/Software tradeoffs are based on the following observations. To obtain a maximal degree of flexibility as much of the functionality as possible is implemented in software on the ARM6. However, due to performance constraints of the ARM6 processor, there is a limit to what can be implemented in software. There are two main factors that play a role in this problem.

- The Tracking and Acquisition process has to be implemented in software because the algorithm used to perform tracking and acquisition may be modified depending on the environment in which the pager system is intended to operate.
- The Correlator and Noise Estimator process is not included in software because the input rate for the Correlator and 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<sup>15</sup> shows that the implementation of Correlator and Noise Estimator process in software leaves insufficient time to perform tracking and acquisition in between every two symbols.

After merging the processes one obtains the system shown in Fig. 16. Each of the merged processes can now be implemented on a separate target processor by the appropriate compiler. The communication between the merged processes is still done via primitive ports and channels.

2) *Communication Mechanism Selection*: After the partitioning of the system has been verified by simulation and before the actual implementation takes place, the designer may choose to refine the communication mechanism between the processors. This can be achieved by expliciting the behavior of the channels between the processors.

<sup>13</sup>The sample rate is 4 complex samples/chip. The input rate for the CMF process is 8 complex samples/chip but it processes even and odd samples concurrently which leaves two cycles available for the processing of each sample.

<sup>14</sup>The sample rate is 1 complex sample/chip because the Correlator and Noise Estimator process selects only one of the 4 complex samples/chip at its input.

<sup>15</sup>This implies the availability of a description in C for the Correlator and Noise Estimator process (possibly derived from the DFL description with DFL2C).

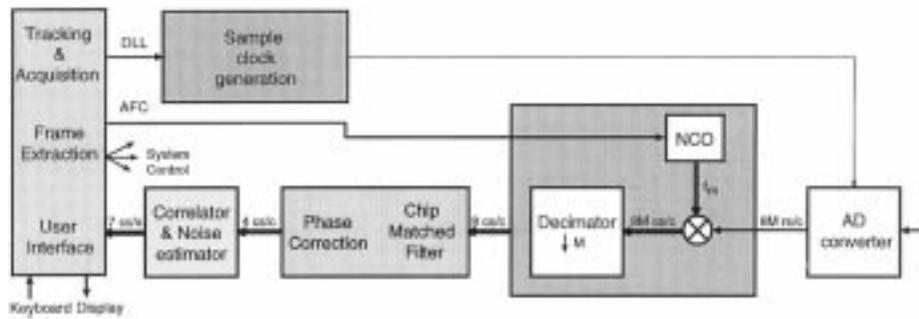


Fig. 16. The pager design after merging of the partitions.

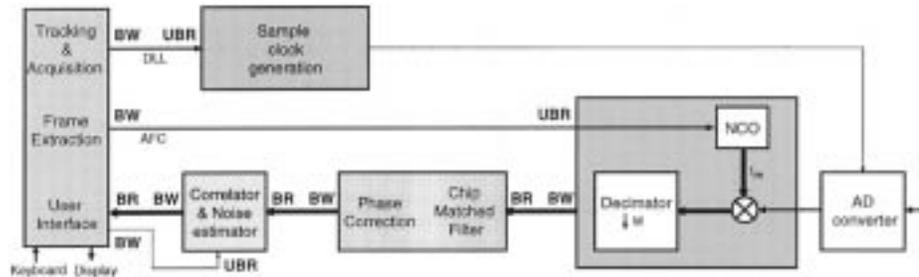


Fig. 17. The pager design with Blocked/UnBlocked Read/Write communication.

In the running example, the processors can, in principle, operate concurrently because each processor has its own thread of control. By refining the RPC based communication scheme we can pipeline the processors: all processors operate concurrently and at I/O points they synchronize. This refined communication scheme is called *Blocked/UnBlocked Read/Write communication*. Fig. 17 shows the pager with the refined communication mechanism. The inputs and outputs of the processors have been labeled with BW for Blocked Write, BR for Blocked Read, and UBR for UnBlocked Read.

BW-BR communication guarantees that no data is ever lost. When the writing process has data available, it will signal that to the reading process. If the reading process is at that moment not ready to receive the data (because it is still processing the previous data), the writing process will block until the reading process is ready to communicate. Alternatively, if the reading process needs new data, it will signal that to the writing process. If the writing process is at that moment not ready to send the data (because it is still computing the data), the reading process will block until the writing process is ready to communicate. The BW-BR scheme is used in the main signal path.

A BW-BR scheme, however, is not used for the parameter and mode setting for the main signal path. If an accelerator uses BR to read a parameter value it will be blocked until the parameter is provided. Since the parameter setting is done in software, this will slow down the computations in the main signal path considerably. Therefore parameter setting is done via a BW-UBR scheme. This makes sure that every parameter change is read by the accelerators, but it leaves it up to the accelerator to decide when to read the parameter.

As explained in Section III-C, in the CoWare design environment the refinement of the communication mechanism is performed by making use of a hierarchical channel. A hierarchical channel replaces a primitive channel by a process that describes how communication over that channel is carried out.

The introduction of BW-BR communication is shown in detail for the CMF and Phase Correction and Correlator and Noise Estimator process in Fig. 18. The BWBR channel contains an autonomous thread and a slave thread that communicate with each other using the mechanism provided by SYMPHONY (see Section IV-A5). The autonomous thread and the slave thread communicate via a channel declared in the context. When both threads are ready to communicate, they synchronize. The rendezvous communication protocol implemented by the send and receive procedures takes care of the blocking of the slave thread and autonomous thread in the BWBR channel. The RPC communication between the BWBR channel and the CMF and Phase Correction and the Correlator and Noise Estimator processes, ensures that these processes are blocked until the data is really passed inside the BWBR channel.

In the case of Blocked-Write, UnBlocked Read communication, slightly different versions of the send and receive procedures are used in the hierarchical channel.

### C. Implementation of the Pager

After the newly introduced communication mechanism has been verified by simulation, each process has to be synthesized on its assigned target processor. In Section VI-C1 we discuss the hardware implementation of some of the processes in the pager. In Section VI-C2 we study the implementation of the software part of the pager.

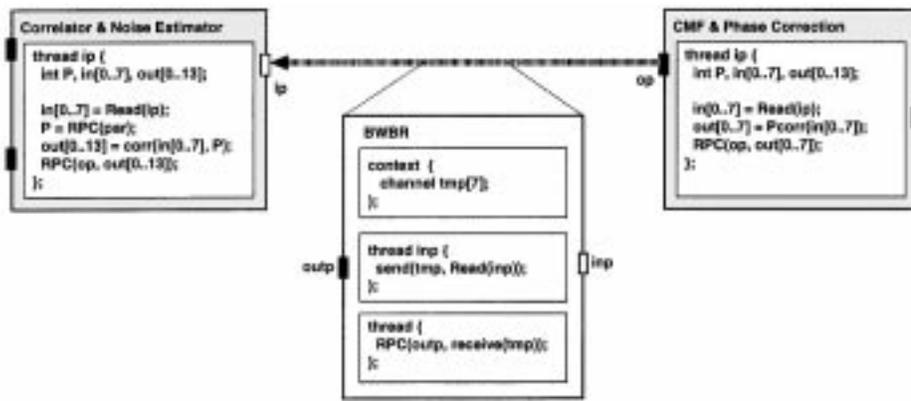


Fig. 18. Introduction of Blocked Write-Blocked Read communication. The channel between ip and op is refined into the hierarchical BWBR channel.

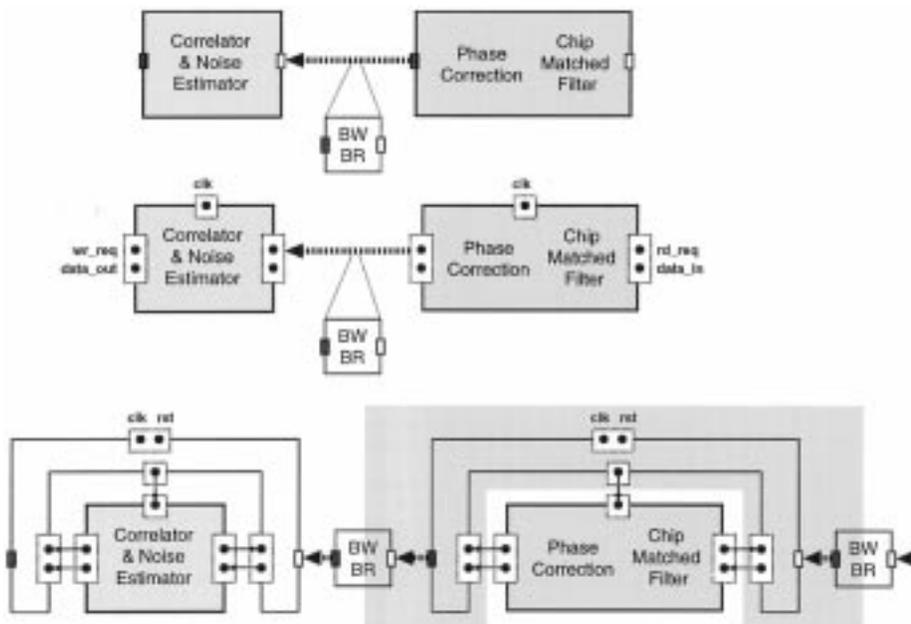


Fig. 19. Implementation in hardware of the Correlation and Noise Estimator process and the merged Phase Correction and CMF process.

1) *Implementation of a Process in Hardware:* Fig. 19 illustrates the pure hardware implementation for the Correlator and Noise Estimator process and the merged Phase Correction and CMF process. This hardware implementation for the paper consists of three distinct steps.

- 1) The (merged) DFL processes are synthesized by the Cathedral silicon compiler. The compiler generates processors of which each input and output is of the master type. In addition, the I/O of the processor have a fixed protocol consisting of 2 terminals ( $\{rd, wr\}_{req}$ ,  $data_{\{in, out\}}$ ). These processors are shown in the middle of Fig. 19.
- 2) Each processor is encapsulated to make it consistent with the specification in which the DFL processes have slave inputs. The encapsulation includes clock gating circuitry to control the activity of the proces-

sor. The encapsulated processors are shown at the bottom of Fig. 19. As can be observed the input ports of the encapsulated processors are now of the slave type.

- 3) The encapsulation blocks and the BWBR channel can now be merged. The merged VHDL code is compiled with SYNOPSIS' Design Compiler. Fig. 20 shows the gate level implementation that corresponds with the encapsulation of the merged Phase Correction and CMF processor (the gray area in Fig. 19).

2) *Implementation of a Process in Software:* To simplify the discussion we will only look at the transfer of the 14 correlation values to the Tracking and Acquisition process, and the setting of a parameter value.

The hardware interface and the software I/O device driver is generated automatically with SYMPHONY (Section IV-B). To generate these interfaces SYMPHONY analyzes the

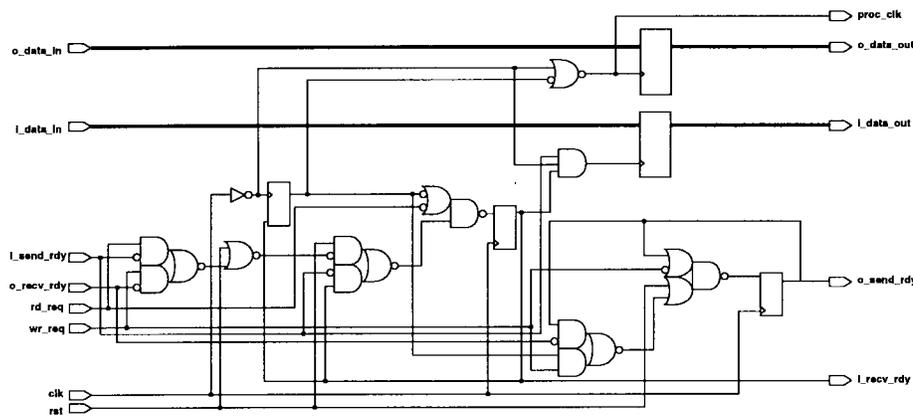


Fig. 20. Gate level net list of the CB and the encapsulation.

ports of the software process. For each of these ports, SYMPHONY scans the library of I/O scenarios for an applicable scenario. The user is asked to select the most appropriate scenario among the applicable ones. SYMPHONY then combines the selected scenarios into a software I/O device driver and a hardware interface.

In the example there are two ports to be implemented:

- `bool[32][14] corr`: `inslave` is used to transfer the correlation values. This is a port of type `inslave`, that transports an array of 14 bitvectors of size 32.
- `bool corr_par`: `outmaster` is used to set a parameter of the correlation block. This is a port of type `outmaster`, that transports a boolean value.

For the `corr` port SYMPHONY proposes the scenario depicted in Fig. 21. The memory port of the ARM will be used to transfer the correlation values and the `fiq` port of the processor will be used to initiate the transfer. The I/O scenario describes what blocks need to be inserted in software and hardware to realize this kind of communication. In total two hardware and two software blocks are required to implement the communication over the `mem` port.

- **Unpack**. The memory port of the ARM is obviously not wide enough to transfer the 14 correlation values in parallel. Therefore, the scenario will sequentialize the transfer. The 14 correlation values will be stored internally in the `Unpack` block. The `Unpack` block activates the `fiq` port, using an RPC on its `go` port. Activating the `fiq` port of the hardware model, has as a consequence that an RPC is issued on the interrupt port of the software model. This port is connected to the `Pack` block.
- **Pack**, when activated by the `fiq` port of the software model, will retrieve the 14 correlation values by issuing consecutive RPC's to the different ports of the demultiplexer `DM1`. When the `Pack` block has retrieved all 14 values, it packs them in an array and activates the original software application code.
- **DM1**. The data transfer is implemented through memory mapped I/O. Therefore, when selecting this I/O scenario, the user should decide on the address that will be used for the transfer. When one of the input ports of the `DM1` block is activated an RPC to the

memory port will be performed with an address that corresponds to the activated input port.

- **M1**. A memory access to the `mem` port in the software model, results in an RPC being issued on the `mem` port of the hardware model. The address of the memory access is given to the multiplexer `M1`. In the multiplexer, the address will be decoded and the corresponding output port will be activated to retrieve the correlation value that was stored locally in hardware in the `Unpack` block.

All these blocks are described in a generic way in a library where they can be retrieved and customized by SYMPHONY.

The solution for the `corr_par` port is much simpler. Since it is an `outmaster` port it can directly be mapped on the memory port. However, since the memory port is already used, an extra multiplexer `M2` and demultiplexer `DM2` are required as shown in Fig. 21.

Finally, the memory port is also used for the memory accesses to the program ROM and data RAM blocks. Therefore a third multiplexer `M3`, needs to be added on the hardware side.<sup>16</sup>

Before going on with the implementation path, all processes that were added by SYMPHONY are merged. The hardware interfaces are merged into one hardware interface block that can then be implemented with RT-level synthesis tools. The I/O device driver processes are merged with the original SW application code. As a consequence of the inlining, the complete tracking and acquisition slave thread ends up in the interrupt routine. Whenever new correlation values are ready, the main software thread is interrupted to run the tracking and acquisition algorithm. After that interrupt is processed, the main thread resumes.

After implementation, the layout of Fig. 2(c) was obtained. After processing, the chip occupies 47 square millimeter in  $0.6 \mu$  VLSI technology. It consists of 340 000 transistors, operates at 40 MHz and, at that frequency, consumes less than 1 W for a supply voltage of 5 V.

<sup>16</sup>The RAM and ROM can either be on-chip (as in Fig. 21) or they can be off-chip.

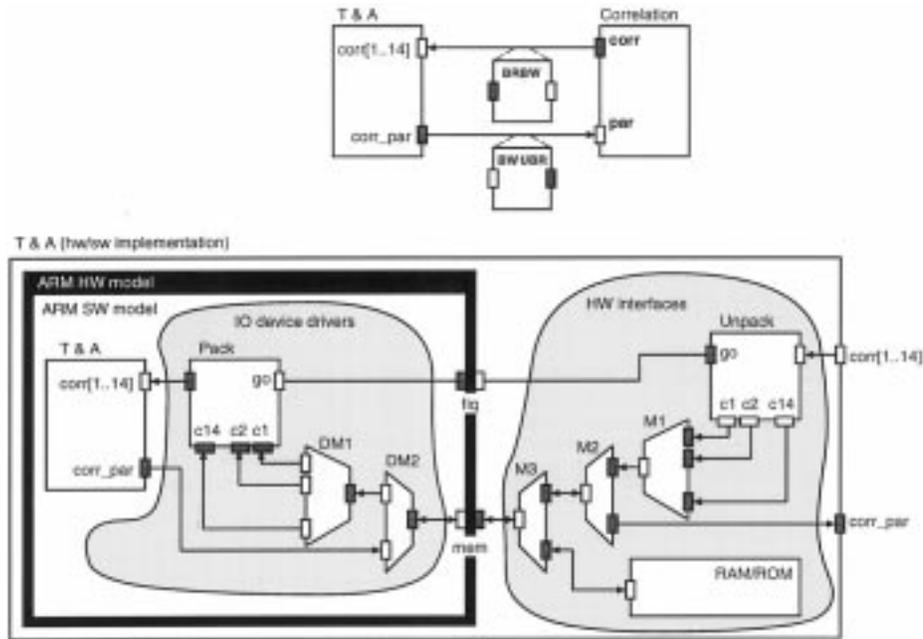


Fig. 21. I/O scenario for passing the correlation values from the hardware accelerator to the application software that runs on the ARM core.

## VII. CONCLUSION

In this contribution we investigated the impact of telecommunication systems and VLSI technology on the design technology needs. Telecommunication systems are heterogeneous as well in terms of specification as in terms of implementation. The system design process must bridge the gap between the heterogeneous functional specification and the heterogeneous implementation of a telecommunication system. Today's component compilers allow to synthesize and program all the components once the global system architecture is defined. What is needed are the models and tools to refine the functional specifications into the detailed architecture: the definition and allocation of the components and their communication and synchronization.

CoWare is a system design environment which supports specification of heterogeneous systems and supports the systematic refinement of the specification to a heterogeneous implementation. The CoWare data model can be used to represent the system throughout the complete design process and in this way provides a formal and executable specification hand-over between design teams. The CoWare data model supports reuse of existing designs and design environments and enables a design for reuse strategy by maintaining a strict separation between functional and communication behavior.

The CoWare data model is implementation oriented. It is designed to allow the description of any kind of system because it has the basic primitives used in hardware and software implementations of systems. The drawback of this design decision is that the data model sometimes lacks in expressiveness. The expressive power of the CoWare data model can be improved in two ways. By relieving a number of restrictions imposed in Sections III-A and III-C,

one can use multipoint channels, function call semantics, or dataflow semantics. Each of these extensions can be automatically translated into the basic data model. The expressive power can also be improved by extending the host language encapsulations. An example of such an extension is the send/receive communication mechanism for intraprocess (interthread) communication in the C and VHDL host language encapsulations (see Section IV).

One of the main focal points of the design environment is the support for the implementation of interprocessor communication. More in particular, the SYMPHONY toolbox was developed for generating the necessary hardware and software to implement communication between the software that runs on a programmable processor and the hardware next to it.

The CoWare design environment is an open environment making it possible for the designer to intervene at any point to change parts of the design.

The implementation of the CoWare design environment is ongoing. Furthermore, some aspects that were neglected in this contribution require further investigation. The use of existing component compilers restricts the system specification. For example, DFL/Cathedral does not allow multiple threads inside a process. These compilers have to be extended to deal with multiple threads. Also, the verification of the functional and timing behavior of complete telecommunication systems is a problem.

Simulation as described in this contribution allows to verify the functional behavior of the system at several abstraction levels. However, as the level at which simulation takes place is lowered, the required CPU time becomes excessive. Verification of the correct real-time behavior of feedback loops such as the tracking and acquisition requires simulation of the complete system over tens of seconds

(real-time). At the implementation level this corresponds to simulations that run for weeks. At higher abstraction levels the simulation times are less excessive. However, when the processes in the system run concurrently (e.g., after pipelining the pager system), the real-time behavior of the system depends on the relative execution speeds of the concurrent processes. These are only known at the implementation level. At higher levels, no conclusions about the correct real-time behavior of the system can be drawn.

When the communication mechanism of a system is refined, for example by pipelining the main DSP path in the pager design, the timing behavior of the system may be radically altered. This might introduce deadlock or starvation of processes and hence analysis of the timing behavior of systems to detect such malfunctions becomes an important issue.

Hence, there is a very hard need for bottom-up formal verification tools both for functionality and timing. Symbolic verification methods, such as signal flow graph tracing [63] and code transformation verification [64], have been shown to be very efficient for the DSP parts. However, a lot of effort is still required to extend these methods to deal with the non-DSP parts of a telecommunication system.

#### ACKNOWLEDGMENT

Much of this work would have been impossible without the cooperation of many colleagues. S. De Troch implemented the UNIX IPC communication for the VHDL simulator, and the pager chip was designed by V. Derudere, B. Gyselinckx, L. Philips, B. Vanhoof, J. Vanhoof, S. Vernalde, and M. Wouters. B. Gyselinckx was especially helpful and patient in explaining the details of the design. This work was supported by A. Demarée, X. Gao, P. Pokorný, I. Makovicky, B. Vanthournout, and S. Samel.

#### REFERENCES

- [1] J. van Meerbergen *et al.*, "PHIDEO: High-level synthesis of high throughput applications," *J. VLSI Signal Process.*, vol. 9, no. 1–2, pp. 89–104, Jan. 1995.
- [2] J. Buck *et al.*, "The token flow model," in *Proc. Data Flow Workshop*, Hamilton Island, Australia, May 1992.
- [3] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Computers*, Jan. 1987.
- [4] ———, "Synchronous data flow," *Proc. IEEE*, vol. 75, Sept. 1987.
- [5] P. N. Hilfinger, J. Rabaey, D. Genin, C. Scheers, and H. De Man, "DSP specification using the SILAGE language," in *Proc. Int. Conf. on Acoust., Speech Signal Process.*, Albuquerque, NM, Apr. 1990, pp. 1057–1060.
- [6] P. Willekens *et al.*, "Algorithm specification in DSP station using data flow language," *DSP Applicat.*, vol. 3, no. 1, pp. 8–16, Jan. 1994.
- [7] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language Lustre," in *Proc. IEEE*, vol. 79, pp. 1305–1320, Sept. 1991.
- [8] P. Chou *et al.*, "Software scheduling in the co-synthesis of reactive real-time systems," in *Proc. 31st Design Automat. Conf.*, June 1994.
- [9] M. Cornero, F. Thoen, G. Goossens, and H. De Man, "Software synthesis for real-time information processing systems," in *Code Generation for Embedded Processors*, P. Marwedel and G. Goossens, Eds. Boston: Kluwer, 1995, pp. 260–296.

- [10] E. Verhulst, "Meeting the parallel DSP challenge with the real-time Virtuoso programming system," *DSP Applicat.*, vol. 3, no. 1, pp. 41–50, Jan. 1994.
- [11] M. Van Canneyt, "Specification, simulation and implementation of a GSM speech codec with DSP station," *DSP and Multimedia Technol.*, vol. 3, no. 5, pp. 6–15, May 1994.
- [12] J. T. Buck *et al.*, "PTOLEMY: A framework for simulating and prototyping heterogeneous systems," *Int. J. Computer Simulation*, Jan. 1994.
- [13] R. Lauwereins *et al.*, "GRAPE-II: A system level prototyping environment for DSP applications," *IEEE Computer*, pp. 35–43, Feb. 1995.
- [14] M. S. Rafie *et al.*, "Rapid design and prototyping of a direct sequence spread-spectrum ASIC over a wireless link," *DSP and Multimedia Technol.*, vol. 3, no. 6, pp. 6–12, June 1994.
- [15] B. Sklar, *Digital Communications: Fundamentals and Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1988.
- [16] D. Gajski *et al.*, *Specification and Design of Embedded Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1994.
- [17] D. Harel, "A visual formalism for complex systems," *Sci. Computer Programming*, no. 8, pp. 231–274, 1987.
- [18] S. Narayan, F. Wahid, and D. D. Gajski, "System specification with the SpecCharts language," *IEEE Design and Test of Computers*, pp. 6–13, Dec. 1992.
- [19] K. O'Brien, T. Ben Ismail, and A. A. Jerraya, "A flexible communication modeling paradigm for system-level synthesis," in *Proc. Int. Workshop on Hardware-Software Co-Design*, Cambridge, MA, Oct. 1993.
- [20] M. Engels *et al.*, "Design of a processing board for a programmable multi-VSP system," *J. VLSI Signal Process.*, vol. 5, pp. 171–184, 1993.
- [21] S. Note *et al.*, "Paradigm-RP: A system for rapid prototyping of real-time DSP applications," *DSP Applicat.*, vol. 3, no. 1, pp. 17–23, Jan. 1994.
- [22] G. Goossens, I. Bolsens, B. Lin, and F. Catthoor, "Design of heterogeneous IC's for mobile and personal communication systems," in *Proc. IEEE Int. Conf. on Computer-Aided Design, ICCAD'94*, San Jose, CA, Nov. 1994, pp. 524–531.
- [23] A. Kalavade and E. A. Lee, "A hardware-software codesign methodology for DSP applications," *IEEE Design and Test of Computers*, pp. 16–28, Sept. 1993.
- [24] I. Bolsens, K. Van Rompaey, and H. De Man, "User requirements for designing complex systems on silicon," in *VLSI Signal Processing*, vol. 7, J. Rabaey, P. Chau, and J. Eldon, Eds. New York: IEEE, 1994, pp. 63–72.
- [25] F. Catthoor, J. Rabaey, G. Goossens, J. van Meerbergen, R. Jain, H. De Man, and J. Vandewalle, "Architectural strategies for an application-specific synchronous multi-processor environment," *IEEE Trans. Acoust., Speech Signal Process.*, vol. 36, pp. 265–284, Feb. 1988.
- [26] V. Zivojnovic *et al.*, "DSP-Stone: A DSP-oriented benchmarking methodology," in *Proc. ICSPAT'94*, Dallas, TX, Oct. 1994.
- [27] J. Vanhoof *et al.*, *High-Level Synthesis for Real-Time Digital Signal Processing*. Boston: Kluwer, 1993.
- [28] P. Paulin, "DSP design tool requirements for embedded systems: A telecommunications industrial perspective," *J. VLSI Signal Process.*, vol. 9, pp. 23–47, Jan. 1995.
- [29] A. Fauth, J. Van Praet, and M. Freericks, "Describing instruction set processors using nML," in *Proc. Europe. Design and Test Conf., ED&TC 1995*, Paris, France, Mar. 1995, pp. 503–507.
- [30] D. Lanneer, J. Van Praet, K. Schoofs, A. Geurts, W. Kifli, F. Thoen, and G. Goossens, "CHESS, retargetable code generation for embedded processors," in *Code Generation for Embedded Processors*, P. Marwedel and G. Goossens, Eds. Boston: Kluwer, 1995.
- [31] M. Strik *et al.*, "Efficient code generation for in-house DSP-cores," in *Proc. Europe. Design and Test Conf., ED&TC 1995*, Paris, France, Mar. 1995, pp. 244–249.
- [32] L. Philips, I. Bolsens, and H. De Man, "A programmable CDMA IF transceiver ASIC for wireless communications," in *Proc. Custom Integrated Circ. Conf., CICC'95*, Santa Clara, CA, May 1–4, 1995.
- [33] S. Note, W. Geurts, F. Catthoor, and H. De Man, "Cathedral III: Architecture driven high-level synthesis for high throughput DSP applications," in *Proc. 28th ACM/IEEE Design Automat. Conf., DAC'91*, San Francisco, CA, June 1991, pp. 597–602.

- [34] F. Balasa, F. Catthoor, and H. De Man, "Data-flow driven memory allocation for multi-dimensional signal processing systems," in *Proc. IEEE Int. Conf. on Computer-Aided Design, ICCAD'94*, San José, CA, Nov. 1994, pp. 31–34.
- [35] F. Catthoor, "Design for high performance dsp systems," in *Tutorial of IEEE Int. Symp. on Circ. Syst.*, Atlanta, May 1996.
- [36] P. Lippens, J. van Meerbergen, and W. Verhaegh, "Allocation of multiport memories for hierarchical data streams," in *Proc. IEEE Int. Conf. on Computer-Aided Design, ICCAD'93*, Santa Clara, CA, Nov. 1993, pp. 728–735.
- [37] W. Geurts, F. Catthoor, and H. De Man, "Time constrained allocation and assignment techniques for high throughput signal processing," in *Proc. 29th ACM/IEEE Design Automat. Conf., DAC'92*, Los Angeles, CA, June 1992, pp. 124–127.
- [38] W. Geurts, "Memory and data-path mapping for image and video applications," in *Application-Driven Architecture Synthesis*. Boston: Kluwer, 1993, pp. 143–166.
- [39] D. S. Rao, "Partitioning by regularity extraction," in *Proc. 29th ACM/IEEE Design Automat. Conf., DAC'92*, Anaheim, CA, June 1992, pp. 235–238.
- [40] S. Vernalde, P. Schaumont, I. Bolsens, and H. De Man, "Synthesis of high throughput DSP ASIC's using application specific datapaths," *DSP and Multimedia Technol.*, vol. 3, no. 6, pp. 13–21, June 1994.
- [41] W. Geurts, F. Catthoor, and H. De Man, "Quadratic zero-one programming based synthesis of application-specific data paths," *IEEE Trans. Computer-Aided Design of Integrated Circ. Syst.*, vol. 14, no. 1, pp. 1–11, Jan. 1995.
- [42] J. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, "Fast prototyping of datapath-intensive architectures," *IEEE Design and Test of Computers*, vol. 8, no. 6, pp. 40–51, June 1991.
- [43] E. Tsern and T. Meng, "A 1.2 mW video-rate 2D color subband decoder," in *Dig. Techn. Papers, Int. Solid State Circ. Conf.*, Feb. 1995, pp. 290–291.
- [44] D. Singh, J. Rabaey, M. Pedram, F. Catthoor, S. Rajgopal, N. Seghal, and T. Mozdzen, "Power conscious CAD tools and methodologies: A perspective," in Special Issue on Low Power Electronics, *Proc. IEEE*, vol. 83, pp. 570–594, Apr. 1995.
- [45] M. Chiodo *et al.*, "Hardware-software codesign of embedded systems," *IEEE Micro*, vol. 14, no. 4, Aug. 1994.
- [46] G. Boriello, "A new interface specification methodology and its application to transducer synthesis," Ph.D. dissertation, Univ. Calif. Berkeley, 1988.
- [47] B. Lin and S. Vercauteren, "Synthesis of concurrent system interface modules with automatic protocol conversion generation," in *Proc. IEEE Int. Conf. on Computer-Aided Design, ICCAD'94*, San Jose, CA, Nov. 1994, pp. 101–108.
- [48] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, pp. 666–677, Aug. 1978.
- [49] P. Chou *et al.*, "Synthesis of the hardware/software interface in microcontroller-based systems," in *Proc. IEEE Int. Conf. on Computer-Aided Design, ICCAD'92*, Nov. 1992, pp. 488–495.
- [50] J. S. Sun *et al.*, "Design of system level interfaces," in *Proc. IEEE Int. Conf. on Computer-Aided Design, ICCAD'92*, Nov. 1992, pp. 478–481.
- [51] W. R. Stevens, *UNIX Network Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1992.
- [52] D. Knapp, T. Ly, D. MacMillen, and R. Miller, "Behavioral synthesis methodology for hdl-based specification and validation," in *Proc. 32nd ACM/IEEE Design Automat. Conf., DAC-95*, San Francisco, CA, June 1995.
- [53] B. Lin and S. Vercauteren, "Synthesis of concurrent system interface modules with automatic protocol conversion generation," in *Proc. IEEE Int. Conf. on Computer-Aided Design, ICCAD'94*, San Jose, CA, Nov. 1994, pp. 101–108.
- [54] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalijs, "The VLSI-programming language Tangram and its translation into handshake circuits," in *Proc. Europe. Design Automat. Conf., EDAC'91*, Mar. 1991, pp. 384–389.
- [55] B. Lin, S. Vercauteren, and H. De Man, "Embedded architecture co-synthesis and system integration," in *Proc. 4th Int. Workshop on Hardware/Software Co-Design*, Pittsburgh, PA, Mar. 1996.
- [56] P. Chou, R. Ortega, and G. Borriello, "The chinook hardware/software co-synthesis system," in *Proc. Int. Syst.-Level Synthesis Symp., ISSS'95*, Cannes, France, Sept. 1995, pp. 22–27.
- [57] J. F. Ready, "VRTX: A real-time operating system for embedded microprocessor applications," *IEEE Micro*, pp. 8–17, Aug. 1986.
- [58] SPECTRON Microsystems, Santa Barbara, CA, *SPOX—The DSP Operating System*, June 1992.
- [59] E. Verhulst, "Virtuoso: Providing submicrosecond context switching on DSP's with a dedicated nano kernel," in *Int. Conf. on Signal Process. Applicat. and Technol.*, Santa Clara, CA, Sept. 1993.
- [60] Etnoteam S. p.A., *EOS—Etnoteam Operating System*.
- [61] S. Vercauteren, B. Lin, and H. De Man, "Constructing application-specific heterogeneous embedded architectures for custom hw/sw applications," in *Proc. 33rd ACM/IEEE Design Automat. Conf., DAC-96*, Las Vegas, NV, June 1996.
- [62] UNIX Syst. Labs. *USL C++ Language System Release 3.0 Library Manual*, 1992.
- [63] M. Genoe, L. Claesen, E. Verlind, F. Proesmans, and H. De Man, "Illustration of the SFG-Tracing multi level behavioral verification methodology," in *Proc. ICCD'91*, Cambridge, MA, Oct. 1991, pp. 338–341.
- [64] H. Samsom, F. Franssen, F. Catthoor, and H. De Man, "Verification of loop transformations for real time signal processing applications," in *VLSI Signal Process. VII*, J. Rabaey, P. Chau, and J. Eldon, Eds. New York: IEEE, 1994, pp. 208–217.



**Ivo Bolsens** received the electrical engineering and Ph.D. degrees from the Katholieke Universiteit Leuven, Belgium, in 1982 and 1989, respectively.

In 1993 he became head of the Applications and Design Technology group at IMEC, where his work focuses on the development and application of new design technology for mobile communication terminals. In 1995 he became Director of IMEC's VLSI Systems and Design Methods division. In 1984 he joined the IMEC Laboratory, where he started doing research on the development of knowledge based verification for VLSI circuits, exploiting methods in the domain of Artificial Intelligence. In 1989 he became responsible for the application and development of the Cathedral-2 and Cathedral-3 architectural synthesis environment. From 1981 to 1984 he was a member of the CAD Group at the ESAT laboratory of the Katholieke Universiteit Leuven, where he worked on the development of an electrical verification program for VLSI circuits and on mixed mode simulation.

Dr. Bolsens received the Darlington Award of the IEEE Circuits and Systems Society in 1986. At the 1991 International Conference on CAD he received a distinguished paper citation, and 1993 he received a best circuit award from the EUROASIC-EDAC conference.



**Hugo J. De Man** (Fellow, IEEE) received the electrical engineering and Ph.D. degrees in applied sciences from the Katholieke Universiteit Leuven, Belgium, in 1964 and 1968, respectively.

Since 1995 he has been a Senior Research Fellow of IMEC, Leuven, Belgium, where he is responsible for research in system design technologies. From 1984 to 1995 he was Vice President of the VLSI Systems Design Group at IMEC, where he worked on design methodologies for integrated systems for telecommunication. He has also been a Visiting Associate Professor (1974–1975) and an ESRO-NASA Postdoctoral Research Fellow (1969–1971) at University of California, Berkeley. He joined the University of Leuven's Laboratory for Physics and Electronics of Semiconductors in 1968, and from 1971 to 1974 he was a Research Associate of the Belgian National Science Foundation at the University of Leuven, where he became Professor in 1974.

Dr. De Man is a corresponding member of the Royal Academy of Sciences, Belgium, and a member of the Royal Flemish Engineering Society (KVIV). He received Best Paper Awards at the ISSCC'73, ESSCIRC'81, ICCD'86, and DAC'89 conferences. In 1987 he received the award for best publication in the *International Journal of Circuit Theory and Applications*.



**Bill Lin** received the B.S., M.S., and Ph.D. degrees in electrical engineering and computer sciences from the University of California, Berkeley, in 1985, 1988, and 1991, respectively.

He joined IMEC in 1992 where he has been heading the the System Control and Communications group researching on various aspects of system design technology. He also held internships at Hewlett-Packard, Hughes Electronics, and Western Digital. He has authored or co-authored more than 70 journal and conference

papers on design automation and VLSI systems

Dr. Lin received a Distinguished Paper citation at the IFIP VLSI conference and at the ICCAD conference in 1989 and 1990, respectively. In 1987 and 1994 he received Best Paper nominations at DAC conferences, and an IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATED SYSTEMS Best Paper Award in 1996. He has served on panels and given invited presentations at the ICCAD conference, the EDTC conference, and the International Workshop on Hardware-Software Codesign. He has also served on the program committees of the DAC and the EDTC conferences.



**Karl Van Rompaey** graduated as an industrial engineer in electronics in 1988.

In 1995 he headed the hardware/software co-design group in IMEC, and in 1996, he co-founded CoWare N.V., where he is now the Director of Engineering. During 1988–1995 he was an R&D engineer at IMEC, Leuven, Belgium, on various aspects of silicon compilation for the Cathedral compilers. In 1988 he was a Software Development Engineer at Agfa Gevaert.

Mr. Van Rompaey won the industry prize for best student and the best thesis award.



**Steven Vercauteren** received the degree in electrical engineering from the Katholieke Universiteit, Leuven, Belgium, in 1992. He is a Ph.D. candidate at the VLSI Systems Design Methodologies Group of the Interuniversity Microelectronics Center (IMEC), Heverlee, Belgium, where he is working on design techniques for heterogeneous embedded systems and hardware/software interfaces.



**Diederik Verkest** received the electrical engineering degree and the Ph.D. degree in applied sciences from the Katholieke Universiteit Leuven, Belgium, in 1987 and 1994, respectively.

In 1987 he joined IMEC Laboratory's VLSI Design Methodology Group, Heverlee, Belgium, as a Research Assistant working on formal design and verification methodologies. Since 1994 he has been working on system design and hardware/software co-design where he is now heading research in the hardware/software design technology group in IMEC.

Dr. Verkest received a Best Paper Nomination at the EURO-DAC'96 Conference.

Dr. Verkest received a Best Paper Nomination at the EURO-DAC'96 Conference.