Design Automation for Embedded Systems, 1, 357–386 (1996)
© 1996 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

CoWare—A Design Environment for Heterogeneous Hardware/Software Systems

D. VERKEST, K. VAN ROMPAEY, I. BOLSENS
IMEC, Kapeldreef 75, B-3001 Leuven, Belgium

H. DE MAN
Katholieke Universiteit Leuven, Belgium

Received August 24, 1995; Accepted February 14, 1996

Abstract. This paper addresses CoWare: an environment for design of heterogeneous systems on chip. These systems are heterogeneous both in terms of specification and implementation. CoWare is based on a communicating processes data-model which supports encapsulation and refinement and makes a strict separation between functional and communication behaviour. Encapsulation enables the reuse of existing specification and design environments (languages, simulators, compilers). Refinement provides for a consistent and integrated path from specification to implementation. The design steps that will be addressed include: system specification, simulation at various abstraction levels, data path synthesis, communication refinement and hardware/software co-design. A spread-spectrum based pager system serves to illuminate the design process in the CoWare environment.

1. Introduction

The functional specification of a complex system involves various specification paradigms. Each of these paradigms has an associated specification language and environment. The signal processing parts of a complex system are best modelled by data-flow graphs with synchronous (SDF [8]) or dynamic (BDF [1]) semantics. These parts can be efficiently described by applicative programming languages such as SILAGE [6] or DFL [18]. The control oriented parts may best be modelled using (hierarchial) co-operating finite state machines which can be specified using, for example, StateCharts [5] or SpecCharts [10]. Other parts, such as the user interface, exhibit reactive semantics.

For each of these paradigms various specification languages and environments are available nowadays. However, to functionally specify a system requires an environment based on a data-model that allows to combine these different specification paradigms and their associated languages and environments. PTOLEMY [2] is a system that provides for such multi-paradigm specification and simulation.

The implementation of a complex system on silicon is also heterogeneous. It consists of a combination of different kind of components such as DSP cores, hardware accelerators, micro controllers, memory and communication blocks, etc. Depending on the required degree of programmability, these systems also contain software components that are implemented on the micro-controller or DSP core. DSP cores are used to perform low-rate signal processing. They can be programmed by making use of a C compiler that accompanies the processor but, due to the poor quality of the code generated by the compiler, are often programmed in assembly language. Hardware accelerators implement the high-rate signal

IPR2024-00476
Petitioner's Exhibit 1007
Page 000001

processing. These components are synthesised from a high-level description (e.g. in Silage or DFL) by accelerator compilers such as CATHEDRAL-3 [11], PHIDEO [17], or HYPER [13]. Micro controllers are used for the implementation of the user interface and/or the control loops and are programmed in C. All these different components communicate with each other through a large number of communication blocks.

For most of the components in a complex system efficient implementation environments are available to date. However, to actually implement the complete system requires an environment that allows to link all the component implementations.

The above observations about the heterogeneous nature of system specifications and implementations have led to the development of the CoWare design environment. The starting point of this development is that both system specification and system implementation consist of an interconnection of heterogeneous components that have to communicate with each other. At the specification level a system consists of communicating *processes*. Each of these processes consists of a behavioral description in some existing specification language (C, VHDL, SILAGE, etc.) each of which has its own existing simulation environment. At the implementation level a system consists of a number of communicating *processors* each of which is generated by a specific existing design environment. In that respect, system simulation is no different from system implementation. For simulation each of the processes in the specification is implemented on a suitable simulator. The key characteristic of heterogeneous system implementations (both for simulation and on silicon) is that each of the components has a fixed interface that is determined by the compiler that generates it. In order to reuse existing specification, simulation, and implementation environments, on an "as is" basis, the design environment has to allow the incorporation of existing components with a fixed interface. In CoWare this *re-use* is achieved through the concept of *encapsulation*. The data-model on which the design environment is based, supports *refinement* which allows a consistent path from system specification to implementation. The design environment supports the refinement of the communication between the processes. The communication between the software components and the hardware components over the fixed hardware interface of a programmable processor is a particular important problem that is dealt with in this environment.

The CoWare environment and the data-model on which it is based are presented in section 2. In section 3 the design of a spread-spectrum based pager system in the CoWare environment is discussed.

2. CoWare: A Data-Model Supporting HW/SW Co-Design

Essential to the philosophy of CoWare is the strict separation between functional behaviour and communication. This allows the introduction of reusable (library) components in a system without a need to modify either the description of the reusable component or the description of the environment in which the component is introduced.

In CoWare, system components are modelled by means of *processes*. Communication between the processes takes place through a behavioral interface, which consists of *ports*. For two processes to be able to communicate, their ports must be connected with a *channel*. The data-model fixes the communication semantics needed when two processes exchange

IPR2024-00476
Petitioner's Exhibit 1007
Page 000002

data and control information. The *communication semantics are built on the concept of the Remote Procedure Call (RPC)*, i.e. one process can trigger the execution of a thread in another process. The channel carries both the control and data information for/from the remote thread. For a motivation of RPC as basic communication mechanism we refer to section 2.1.

The data-model is hierarchically structured and allows to refine a *process*, a *channel*, and a *port* into lower level *processes*. Figure 1 shows the relations between the data-model objects. The semantics of the data-model objects are defined as follows:

- *Processes* are used to describe the behaviour of the system components. A *primitive process* is one whose behaviour is described in a host language, e.g. C, DFL, or VHDL. A *hierarchical process* is defined in terms of other processes and described in the CoWare language. Every process has a unique behavioural interface and at least one implementation. If a process has alternative implementations, then the behaviour of these implementations should be functionally equivalent.
  A primitive process consists of a context and a number of threads. The *context* contains code that is common to all threads in the process. It is used for declaring variables and defining functions that are shared by all threads. As such, the context provides for inter-thread (intra-process) communication.

- *Ports* are objects through which processes communicate. A port can be primitive or hierarchical. Hierarchical ports are used to describe protocol conversions and data formatting. A *primitive port* is used in all other cases; it consists of a *protocol* and a data type parameter.
  Each process has one special, implicit, *construct* port. This slave port is activated exactly once when the system is started.

- *Protocols* define the communication semantics of a port. Protocols can be primitive or hierarchical. A *primitive protocol* is one of *inmaster, outmaster, inoutmaster, inslave, outslave, inoutslave*. Each primitive protocol indicates another way of data transport. The *in, out* or *inout* prefix indicates the direction of the data. The *master, slave* postfix indicates whether the protocol activates an RPC (master) or services and RPC (slave). A primitive protocol does not imply any specific protocol implementation, i.e. it does not fix the timing diagram. For that purpose, one must use hierarchical protocols. Such a *hierarchical protocol* makes use of a number of *terminals* and further consists of a timing diagram that indicates how the logic values on the terminals evolve over time during data transport.

- A *thread* is a single flow of control within a process. A process can contain multiple threads. We distinguish between *slave threads* and *time-loop threads*.
  - *Slave threads* are uniquely associated to slave ports and their code is executed when the slave port is activated. There is one special slave thread which is associated to the *construct* port and can be used to initialise the process.
  - *Time-loop threads* are not associated to any port and their code is executed in an infinite time-loop.

IPR2024-00476
Petitioner's Exhibit 1007
Page 000003

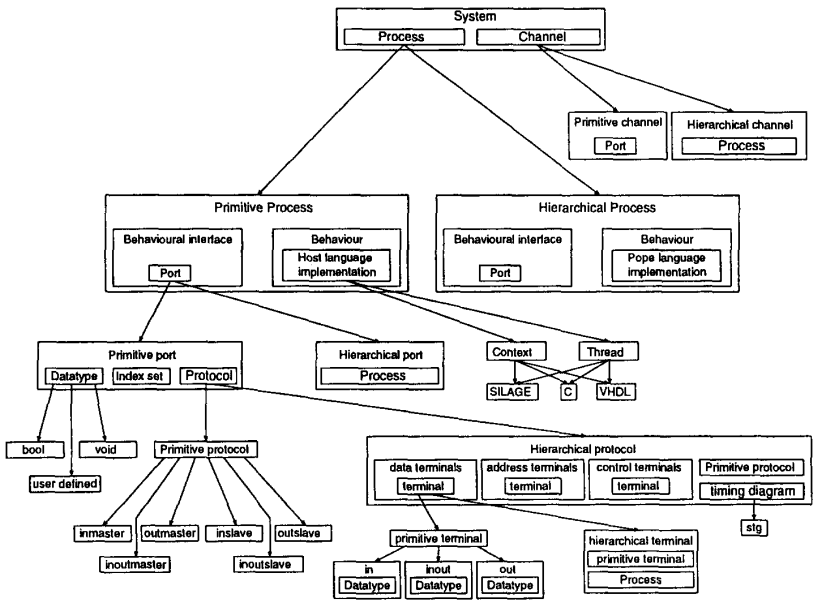


Figure 1. The CoWare data-model objects.

- A *terminal* is one of *in, out* or *inout*, indicating the data direction.
- A *channel* connects a master port to a slave port. Two ports that are connected by a channel can exchange data. A channel can be primitive or hierarchical. A *primitive channel* provides for unbuffered communications. A *hierarchical channel* is a process of which the behavioural interface is fixed by the ports that are connected by the channel. At the *conceptual level*, hierarchical channels can be used to model a communication channel, e.g. bandwidth limitations or noise sources interacting with a wireless communication channel. For *implementation*, a hierarchical channel is used to specify communication buffers, e.g. FIFOs, stacks.

The example in Figure 2 shows the refinement of a very simple system that consists of two processes that communicate over a single channel. At the most abstract level (top of Figure 2) the system is described by the two processes P1 and P2. Process P1 has a master port p1 which is connected by a channel to the slave port p2 process P2. The behaviour of process P1 is specified by a timeloop thread. The behaviour of process P2 is specified by a context and two slave threads connected to the slave ports of the process. Once the system is started, the timeloop thread in process P1 executes continuously. Every time the statement RPC(p1, data) is reached, a remote procedure is started over port p1. The timeloop thread in process P1 is halted, until the RPC is complete. The RPC is serviced by

IPR2024-00476
Petitioner's Exhibit 1007
Page 000004

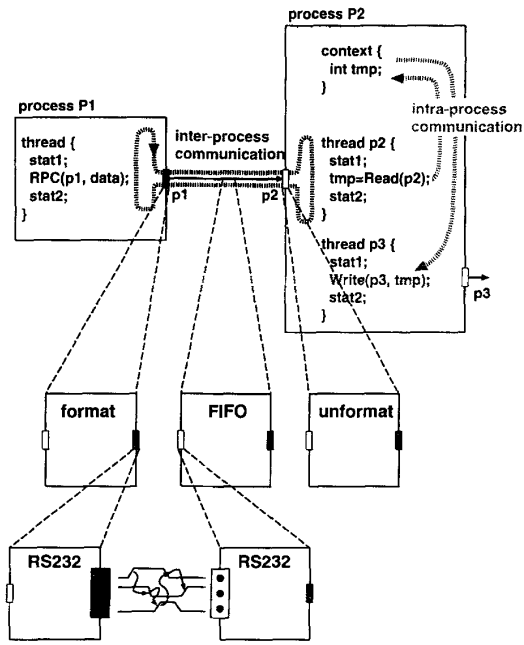


Figure 2. Illustration of the usage of the data-model objects.

slave thread p2 that is connected to port p1 via the channel. In the code of slave thread p2 the data is read from the associated slave port p2 and stored in the shared variable tmp that is declared in the context of process P2. After the last statement (which is not necessary the Read statement) of the slave thread is executed, the timeloop thread in process P1 resumes. Process P2 also contains a second slave thread that services an RPC over slave port p3 and writes the value of the shared variable tmp to some other process.

A first refinement consists of formatting the data that is transported over the channel. This is achieved by making the primitive ports, p1 and p2, hierarchical.A hierarchical port is a process that describes the behaviour that we want to implement in the primitive port. 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 p2 can then use this CRC to determine whether the received data is valid.

A second refinement consists of making the channel hierarchical by specifying its behaviour in a process. In the example the hierarchical channel has a FIFO behaviour. The FIFO process decouples the timeloop 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 calls is no longer determined by the rate at which process P2 can service the RPC calls. The FIFO process takes care of the necessary buffering of data.

IPR2024-00476
Petitioner's Exhibit 1007
Page 000005

A final refinement consists of replacing the ports of the format and FIFO process by hierarchical ports with a hierarchical protocol. In the example of Figure 2, these hierarchical ports implement an RS232 protocol. In the RS232 hierarchical port an RPC call issued in the format process is converted to manipulations of logical values on the terminals according to a timing diagram. When the behaviour of the RS232 hierarchical port is described in the C language, the terminals can be manipulated by the three functions Put, Sample, and Wait. When the behaviour is described in VHDL, the terminals can be manipulated directly.

By using primitive ports and primitive protocols, the designer can concentrate on the functionality of the system while abstracting from terminals, signals, and handshakes. As the design progresses, these abstract objects (ports and protocols) are refined by making them hierarchical: a hierarchical protocol or port describes in a more detailed way the behaviour of the primitive port or protocol. Hierarchical ports are used for protocol conversion (implementation) and data formatting (conceptual level). Hierarchical protocols allow to refine the specification of a protocol with a timing diagram and its associated terminals. Hierarchical channels allow the specification of complex communication behaviour both at the conceptual and implementation level. Hierarchical processes are used to split a complete system into manageable parts.

The example of Figure 2 also nicely illustrates the three communication mechanisms supported by the CoWare environment:

- intra-process (inter-thread) communication via shared variables in the context of the process;
- inter-process (inter-thread) communication with a primitive protocol of which the behaviour is based on RPC semantics;
- inter-process (inter-thread) communication with a hierarchical protocol of which the behaviour is completely determined by the protocol.

2.1. RPC as a Basic Communication Mechanism

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 modelled by making use of RPC.
- **Modular**: functional and communication behaviour of a process are separated.
- **Removable**: a slave thread servicing an RPC can be inlined in the calling process with zero overhead.

The above observations can be summarised in the following statement: *RPC supports design for re-use and re-use of designs.*

IPR2024-00476
Petitioner's Exhibit 1007
Page 000006

Traditionally, the description of a component contains both functional and communication behaviour in an interleaved way. When such a component has to be re-used, 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 behaviour is described by a process that makes use of RPC calls to communicate with the outside world. Such processes can be connected with each other without modifying their description (modularity). Further on in the design process, hierarchical channels, ports and protocols can be used to refine the inter-process communications behaviour. 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 communication blocks that are re-usable: 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 RPCs (see section 2.4). 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.

2.2. *Simulation in the CoWare Environment*

In this hardware/software co-design environment, system simulation is considered as a pure software implementation of the system on a UNIX work station. In Figure 3 the simulation script is shown taking into account the three host languages considered so far. For each host language a path to simulation is required.

In general, building a system simulation consists of the following steps:

1. The DFL processes[1] are converted to C processes using a DFL to C translator [16] (DFL2C in Figure 3). Next, the designer decides what processes to run on what simulator. Normally, all processes described in the same host language will run on a single simulator for that host language. However, distributed simulation is also possible by using more than one simulator for a host language (e.g. the two VHDL branches in Figure 3 are simulated on different VHDL simulators).

2. For efficiency, all processes assigned to the same simulator are merged by a tool called INLINER (see section 2.4). We now have a single, possibly multi-threaded, merged process per simulator.

3. The communication between the single merged C process and the merged VHL process(es) corresponds to the initially specified communication. However, the VHDL simulator(s) and the multi-threaded C library [15] on which the VHDL process(es) and the C process have to run can not, as such, communicate with each other. To make them communicate, UNIX specific communication primitives must be used. This is achieved by making the ports of the merged C and VHDL process(es) hierarchical. The hierarchical ports are available from a library and describe how the VHDL simulator(s) and the multi-threaded C library can communicate using UNIX communication primi-

IPR2024-00476
Petitioner's Exhibit 1007
Page 000007

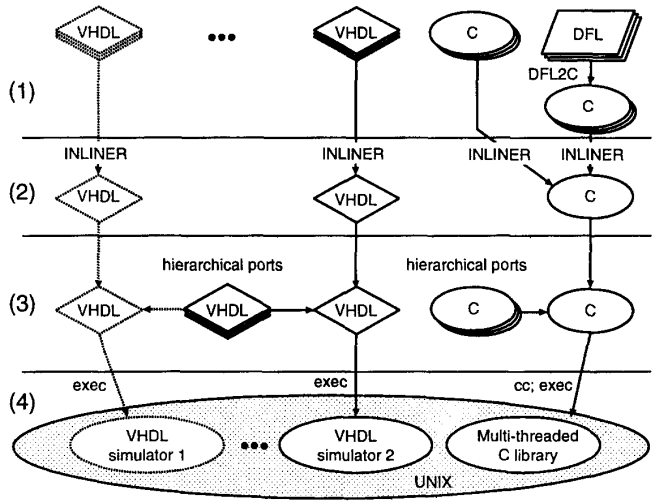


Figure 3. The compilation script for simulation.

tives. The VHDL and C port processes that are introduced are again merged into one VHDL and C process, respectively.

4. Finally, the merged VHDL process(es) are executed on the VHDL simulator(s) and the C code of the C process is compiled, linked with the multi-threaded C library and executed: the system simulation is running.

Input stimuli for the system simulation are generated by C processes that are added to the system description. These stimuli generators e.g. read input data from a file or contain functions that generate input data. In the same way the system outputs and internal signals can be observed by adding monitor processes to the system. These monitor processes e.g. dump data to a file or display them in a window on the work station. In the simulation script these generator and monitor processes are treated in exactly the same way as any other C process.

2.3. Principles of Hardware/Software Co-Design in CoWare

The hardware/software implementation script, shown in Figure 4, does not differ much from the simulation script. The main difference is in the implementation path for C processes, where hardware/software co-design is important, and in the final integration of the processors, where protocol conversion is important.

Before starting the compilation, the processes are merged into clusters each of which is assigned to a compatible processor.

IPR2024-00476
Petitioner's Exhibit 1007
Page 000008

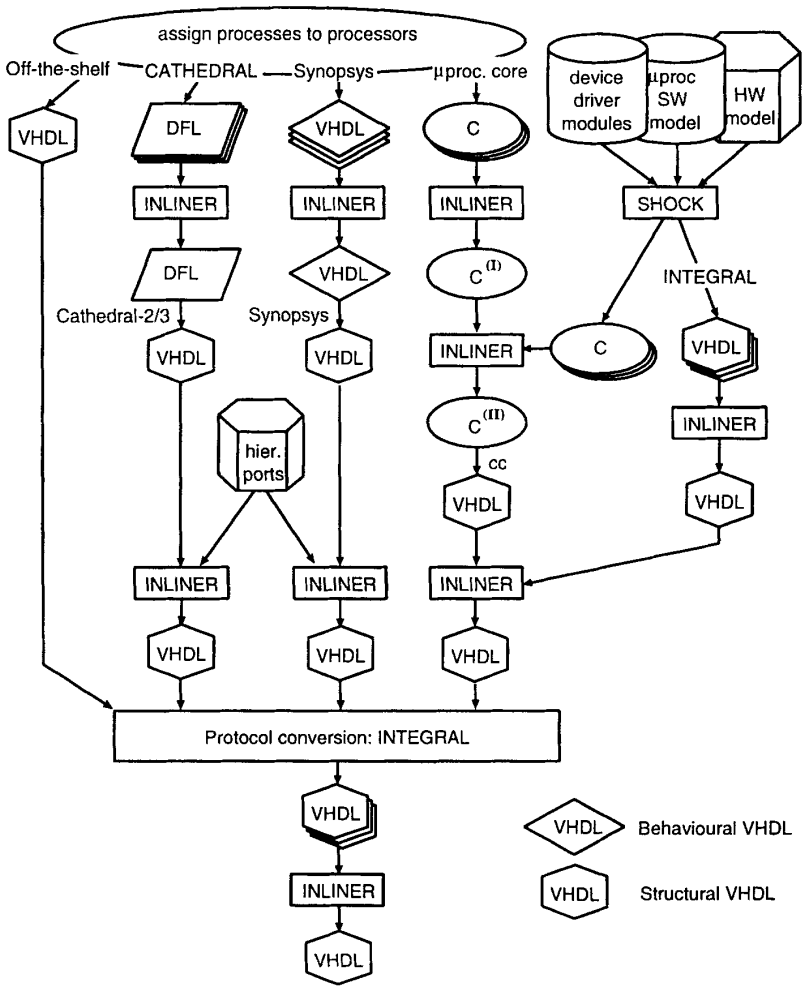


Figure 4. The script for hardware/software implementation.

The path for DFL and VHDL processes[2] is quite similar: all processes assigned to the same processor are merged by the INLINER and that single process is compiled by the appropriate compiler (CATHEDRAL-2/3 and Synopsys, respectively) resulting in a single VHDL process (a net list). To encapsulate the generated processors, the appropriate hierarchical ports are added. The INLINER is then called once more to merge these VHDL processes.

The path for C processes is more complicated. After inlining all C processes assigned to the same processor we would like to compile the result (C process (I) in Figure 4) directly onto the processor core. However, the interface of a processor core is fixed and, usually, does not correspond with the interface required for C process (I) that has to be executed on it. To remedy this a double action has to be taken.

1. A number of C processes (device drivers) need to be added at the software side. The resulting extra C processes are again inlined and the result (C process (II) in Figure 4) is compiled on the processor core. The result is a VHDL process (a ROM with the machine code of the program including the device drivers).

2. Some extra hardware components need to be added at the hardware side of the processor. These extra VHDL processes are then merged with the VHDL process representing the hardware view of the processor core to produce a single VHDL process.

The addition of the extra software and hardware processes is performed by the tools SHOCK and INTEGRAL that are discussed in sections 2.5.

The task remaining at this point is the integration of the VHDL processes. Because the ports that were introduced in the different implementation paths have hierarchical protocols (i.e. terminals and a timing diagram) that are not necessarily compatible, they cannot in general be connected straight away. Protocol conversion will be required in between every two interconnected ports that have an incompatible hierarchical protocol. The protocol conversion tool INTEGRAL [9] is used for this purpose.

After protocol conversion, the complete system consists of processors described in VHDL and all connected ports are compatible. A final call to the INLINER merges all these VHDL processes into a single system processor described in VHDL.

Both the compilation and the implementation path make use of only a small set of tools that will be discussed in the next sections.

2.4. INLINER: A Process Merger

The goal of the INLINER is to merge all processes described in the same host language and assigned to the same target processor.

In the process of merging, all remote procedure calls are inlined: each slave thread is inlined in the code of the master thread that calls it. This "inlining" transformation eliminates the overhead that accompanies the execution of (remote) procedure calls. It reduces the number of threads and therefore the overhead that accompanies the switching between threads. Finally, it allows compilers to optimise the merged process which allows optimisation over the boundaries of the original processes.

IPR2024-00476
Petitioner's Exhibit 1007
Page 000010

The result of merging is a single process that

- contains a single context;
- contains a single construct thread;
- contains one or more time-loop threads;
- possibly contains multiple slave threads through which RPC requests from external processes (assigned to different processors) are serviced;
- possibly contains multiple RPC calls to slave threads in external processes (assigned to different processors).

An example of the INLINER at work is given in Figure 9 in section 3.3.1.

2.5. SHOCK: Software/Hardware Optimizing Co-Design Kit

In this section we discuss the tasks that need to be performed to implement a (software) process on a programmable processor and connect it with the (hardware) processes it communicates with. The environment that accomplishes this is called SHOCK and is schematically depicted in Figure 5. The task of SHOCK is to generate the I/O device drivers and the hardware interfaces. The I/O device drivers link the original C process (1) to the software interface (3) of the processor. The hardware interface makes the link between the hardware interface (2) of the processor and the processes external to the processor. The input to SHOCK consists of:

1. The primitive CoWare process described in C that must be mapped onto the programmable processor.
2. A hardware model of the programmable processor consisting of a CoWare process that formalises the information that is available in the hardware section of the data sheet of the programmable processor. For each programmable processor used in the design team, this hardware model has to be constructed. Once constructed, the hardware model is part of a library from where it can be retrieved by SHOCK. In Figure 5 this hardware model is represented by the rectangle marked "processor HW model" and the ports at the outside of the rectangle's perimeter.
   The hardware model CoWare process is characterised by
   - A behavioural interface that is conforming with the hardware interface of the programmable processor. All ports have hierarchical protcols: they consist of terminals and a timing diagram (including timing constraints) as is shown in Figure 5 for one particular port.
   - An optional[3] VHDL description
     - at net list level. This net list can be used for simulation and can also be included in the final description of the implementation.

IPR2024-00476
Petitioner's Exhibit 1007
Page 000011

- at simulation level. Usually the exact net list of a processor is not available. Instead a simulation model is provided that behaves exactly as the net list in terms of timing, bit-true behaviour, ...

3. A software model of the programmable processor consisting of a CoWare process that formalises the information that is available in the software section of the data sheet of the programmable processor. For each programmable processor used in the design team, this software model has to be constructed. Once constructed, the software model is part of a a library from where it can be retrieved by SHOCK. In Figure 5 this software model is represented by the rectangle marked "processor SW model" and the ports at the inside of the rectangle's perimeter.
SHOCK is based on the observation that programmable processors have a number of common communication methods to get data in or out of the processor (memory mapped, via a co-processor port, etc). SHOCK is an interactive tool, that leaves the decision of which communication method to use to the user. Once the communication method is fixed, SHOCK generates the I/O device drivers for that communication method by combining a number of generic processes. These I/O device drivers are based on a model of the communication method that makes abstraction of the details of the method for the specific processor. The link between the abstracted model of a communication method and the actual implementation on a specific processor is made via the software model of the processor.
The software model is a CoWare process characterised by

   - A behavioural interface that is conforming with the software interface of the programmable processor. All ports have primitive protocols (the software model is more abstract). The software model identifies, for example, what ports can be used as interrupt ports and what their characteristics are (priority of the interrupt, maskable interrupt or not, ...). Some ports that are present in the hardware model, are not found back in the software model. For example, the ARM6 processor has (hardware) ports that indicate the mode of the processor: normal processing, dealing with interrupt, etc. These ports are not directly accessible from the software side and hence do not feature in the software model of the processor.
   - A behavioural 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. The compilation can be done by a target specific compiler or by a retargetable compiler (e.g. CHESS [7]). In the latter case an instruction level description of the processor should be available (e.g. in nML [3]). In the former case such a description is not needed; it is hard linked in the target specific compiler.

   The output of SHOCK is a set of primitive CoWare processes described in C and VHDL.

   - The C processes added by SHOCK are I/O device drivers.
   - The VHDL processes are the hardware interfaces used by the I/O device drivers to cross the hardware/software boundary. The VHDL processes are synthesised

IPR2024-00476
Petitioner's Exhibit 1007
Page 000012

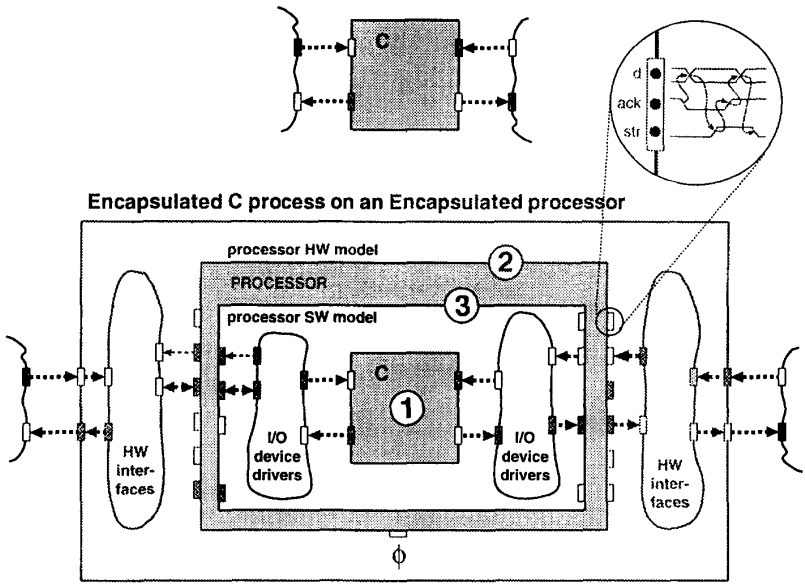


Figure 5. Mapping a C process to a programmable processor. The original C process is shown at the top. At the bottom, we see the result of SHOCK: an encapsulated process on an encapsulated processor. From outside to inside, one recognises: the HW interface to the processor, the processor HW model (marked 2), the actual processor, the processor SW model (marked 3), and finally the original C process (marked 1).

by INTEGRAL [9] starting from information provided by SHOCK. This information consists of the protocols of the processor ports, the protocols of the ports of the original C process, and a high-level behavioural description of the hardware interface.

The hardware/software model of the programmable processor is a representation of the hardware/software boundary that must be crossed to implement the behavioural C process onto a specific programmable processor. The channels which are present in the system specification (at the top of Figure 5) must cross the hardware/software boundary in some way or another. The extra hardware and software added by SHOCK and INTEGRAL achieves just this. The original C process is found back unmodified as the inner rectangle in Figure 5. The outermost rectangle has exactly the same behavioural interface as the original C process defined in the functional specification and, therefore, interfaces to the environment in exactly the same way as the original C process. One could say that INTEGRAL introduces hardware interfaces of which the functionality is exactly the inverse of the software I/O drivers introduced by SHOCK.

IPR2024-00476
Petitioner's Exhibit 1007
Page 000013

3. Co-Design of the Pager Application

In this section we discuss the application of the design trajectory outlined in section 2 to the pager system. We discuss how the functional specification of the pager in Figure 6.a is gradually refined into the mixed hardware/software implementation shown in Figure 6.b.

3.1. Specification of the Pager

Each block in Figure 6 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.* It does not make sense to have a finer grain partitioning: the communication inside each process is very intense, the communication with external processes is limited. Hence, the partitioning of the system is determined by the functional characteristics of the system.

The arrows in between the processes indicate communication via a Remote Procedure Call (RPC) mechanism. Figure 7 shows the RPC communication in detail for *part* of the pager design. The blocks in the figure correspond to the processes from Figure 6.a. The small rectangles on the perimeter of the processes are the ports. The shaded ports are master ports, the others are slave ports.

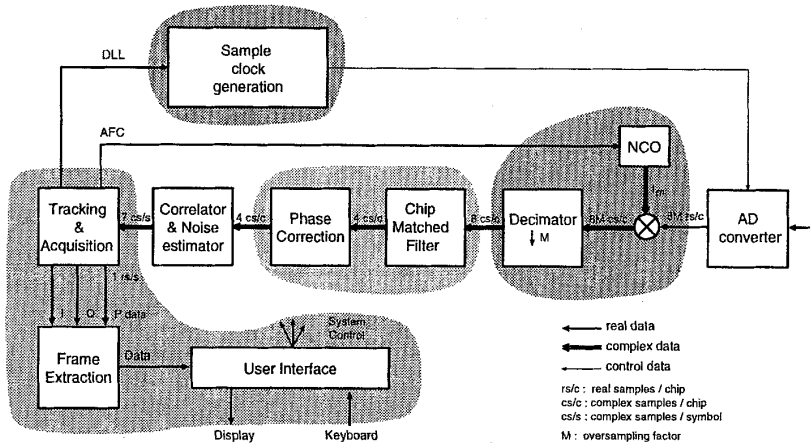
The Sample Clock Generator process contains a time-loop thread. This thread runs continuously. It performs an RPC over its input port ip to the Tracking & Acquisition process to obtain a new value for delta. The time-loop thread of the process adds the delta parameter to some internal variable until a threshold is exceeded. In this way it implements a sawtooth function. When the sawtooth exceeds the threshold an RPC call is issued to the A/D converter process.

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

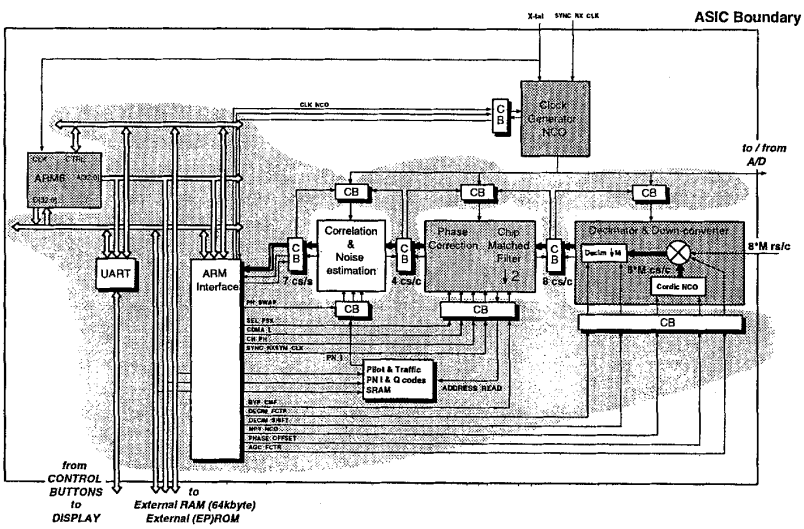
The Correlator & Noise Estimator process contains a slave thread associated with port ip to compute the correlation values. This slave thread is activated when the Phase Correction process writes data to the Correlator & Noise Estimator process (i.e. when the Phase Correction process performs an RPC to the ip port of the Correlator & Noise Estimator process). The slave thread reads in the data and then performs an RPC to the User Interface 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 & Acquisition process via an RPC call on its op port.

The slave thread in the Tracking & 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 & 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

IPR2024-00476
Petitioner's Exhibit 1007
Page 000014



(a)



(b)

Figure 6. (a) Functional specification of a spread-spectrum pager, (b) a possible implementation using an ARM processor-accelerator-memory architecture.

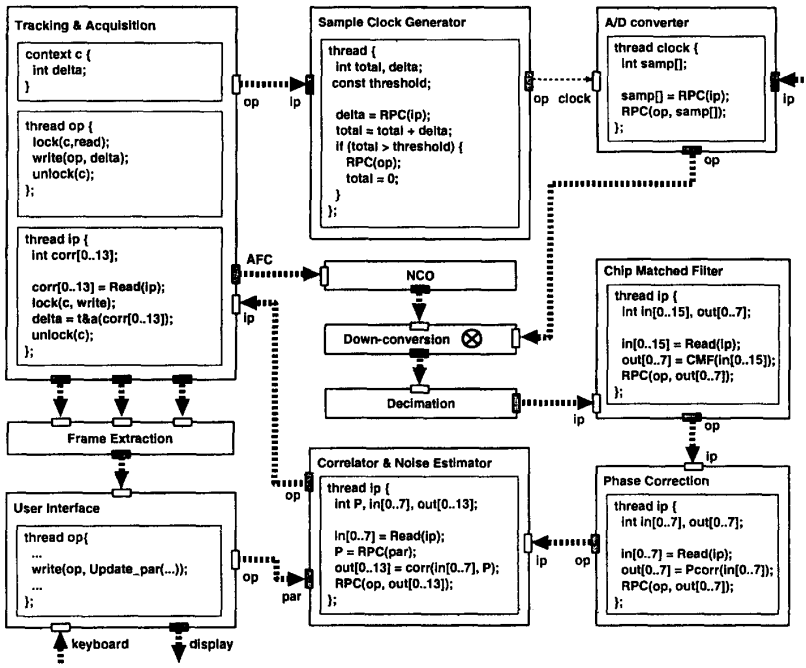


Figure 7. RPC as a basic communication mechanism between processes.

accesses to the variable delta. The lock 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 to 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.[4] The data-flow blocks (NCO, Down-conversion, Decimination, Chip Matched Filter, Phase Correction, Correlator & Noise Estimator, and Sample Clock Generator) are described in Silage/DFL. The control oriented blocks (Tracking & Acquisition, Frame Extraction and User Interface) are described in C. The description of the DSP blocks consists of approximately 900 lines of Silage/DFL code. The description of the control oriented blocks consists of approximately 100 lines of C code.

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 implemented. 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 buildings 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.

IPR2024-00476
Petitioner's Exhibit 1007
Page 000016

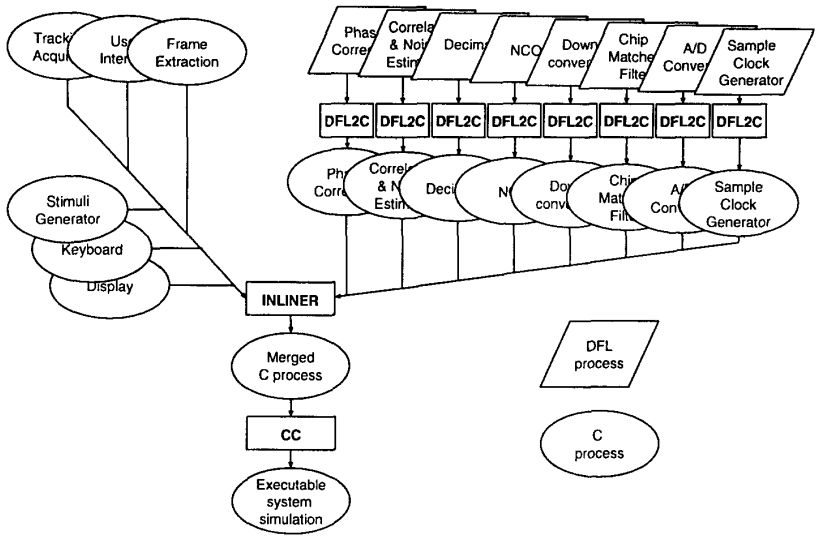


Figure 8. Simulation of the initial pager system specification. At the left-hand side, the C processes from the specification and the stimuli generator and monitor processes are shown. At the right-hand side, the DFL processes from the specification are shown.

3.2. Simulation of the Initial Specification

Simulation is a repeated activity during the design process. After every refinement step, simulation is used to verify the correctness of the refinement step. In section 2.2 the simulation script is explained. Figure 8 applies this script to the pager design of Figures 6.a and 7. All the DFL processes are translated to C processes by DFL2C. These C processes are then merged by the INLINER with the original C processes from the pager system specification and with the stimuli generator and monitor processes. In the pager system, input stimuli for the A/D Converter process are generated by a process that reads the data that we want to page from the keyboard, encodes the data using the algorithms that are implemented in the transmitter, and modifies the encoded data to simulate the distortion of the signal (noise, reflection, Doppler shift, . . .) in a real transmission channel.

3.3. Design Process

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

IPR2024-00476
Petitioner's Exhibit 1007
Page 000017

3.3.1. Partitioning and Mapping

This step determines what processes will be implemented on what target processor. As explained in section 3.1 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 opportune 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. 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 & Noise Estimator process) a description is available in more than one specification language. After partitioning and binding, one obtains a partitioning of the initial specification such that there is a one-to-one binding of merged processes to processors.

In the pager example (Figure 6.a) the following merging and binding takes place:

- The NCO, Down-conversion, and Decimation processes are merged and mapped in hardware onto an application specific DSP CATHEDRAL processor because the sample rate of the merged processes is identical[5] 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.

- The Chip Matched Filter, and Phase Correction processes are merged and mapped onto a CATHEDRAL processor because their sample rates are identical.[6]

- The Correlator & Noise Estimator process is mapped onto a CATHEDRAL processor. It is not merged with the Phase Correction process because it operates at a four times lower frequency.[7]

- The Sample clock generator is mapped onto a CATHEDRAL processor.

- Tracking & Acquisition, Frame Extraction, and User Interface 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. The two main factors that play a role in this problem are

  - The Tracking & Acquisition process has to be implemented in software because the algorithm used to perform tracking and acquisition may be modified depending on the application domain of the pager system.

  - The Correlator & Noise Estimator process is not included in software because the input rate for the Correlator & Noise Estimator is too high to realise 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

IPR2024-00476
Petitioner's Exhibit 1007
Page 000018

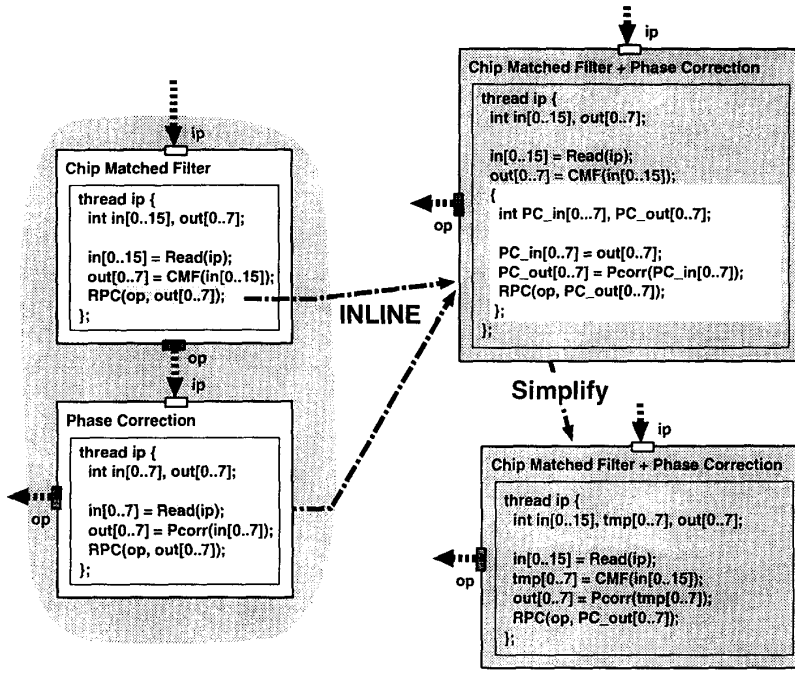


Figure 9. Merging of the Chip Matched Filter and Phase Correction processes.

ARM6[8] shows that the implementation of Correlator & Noise Estimator process in software leaves insufficient time to perform tracking and acquisition in between two symbols.

Figure 9 shows the effect of merging the Chip Matched Filter process and the Phase Correction process. On the right-hand top side the RPC call in the Chip Matched Filter process has been replaced with the code of the slave thread in the Phase Correction process. After simplification one obtains the description of the merged process at the right-hand bottom side of Figure 9. The communication overhead associated with an RPC call has completely vanished. We refer to section 2.4 for details about the merging.

Similarly the other partitions are merged. The result is shown in Figure 10. 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 the RPC mechanism. *At this point a simulation of the partitioned system could be done to verify that the merging step was carried out correctly.*

IPR2024-00476
Petitioner's Exhibit 1007
Page 000019

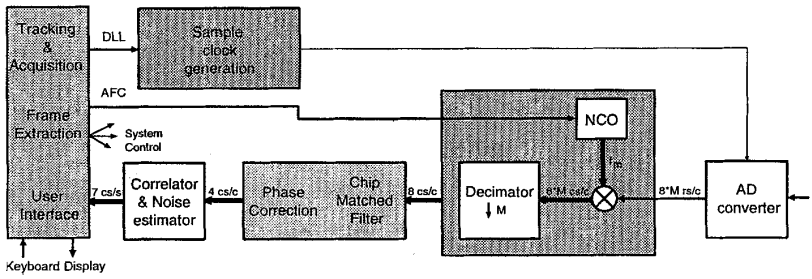


Figure 10. The pager design after merging of the partitions.

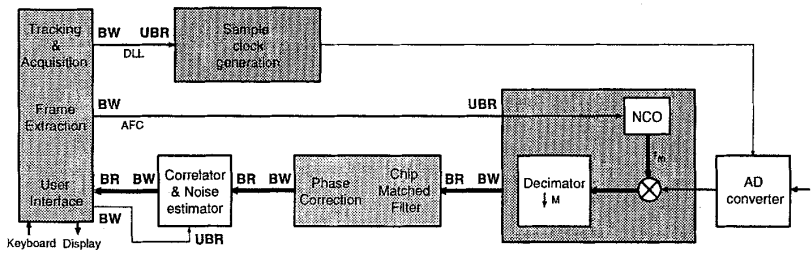


Figure 11. The pager design with Blocked/UnBlocked Read/Write communication.

3.3.2. *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 explicitising the behaviour of the channels between the processors.

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 synchronise. This refined communication scheme is called *Blocked/UnBlocked Read/Write communication.* Figure 11 shows the pager with the refined communication mechanism. The input and outputs of the processors have been labelled 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

IPR2024-00476
Petitioner's Exhibit 1007
Page 000020

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 is 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.

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 hierarchical channel contains more detail than the primitive channel and hence can be considered as a refinement of the primitive channel.

The introduction of BW-BR and BW-UBR communication is shown in detail for the Correlator & Noise Estimator and the Tracking & Acquisition process in Figure 12. The BWBR channel contains a time-loop thread and a slave thread that communicate with each other via the shared variable tmp[0..13] in the context. The slave thread is activated by an RPC from the Correlator & Noise Estimator process and it tries to update the context with new correlation values. The time-loop thread continuously tries to read the correlation values from the context and send them to the output port via an RPC, in this way activating the Tracking & Acquisition process that is attached to that output. The blocking character of the communication is taken care of by the use of a binary semaphore rw. This guarantees that the input thread will block until the previous data has been read by the time-loop thread (no data is overwritten before it has been read), and that the time-loop thread will block until new data is available (no data is read twice). When the input slave thread is blocked, the Correlator & Noise Estimator process that requested its service via an RPC is also blocked because the RPC will only return after the slave thread has completed. When the time-loop thread is blocked, there are no RPC requests to the Tracking & Acquisition process, so that process is blocked automatically.

In the case of Blocked Write, UnBlocked Read communication, the code for the time-loop thread is slightly modified. The thread always sends the value stored in the context, without checking whether it is updated. The same value can be sent more than once, but the thread will never be blocked. The input slave thread is identical to the BWBR case, and will block until the data has been read.

In both cases, locking and unlocking of the context is required to avoid concurrent accesses to the shared variable in the context and, as such, has nothing to do with the blocking character of the communication.

3.3.3. Simulation of the Communication Mechanism

The refinement of the system communication to a Blocked/UnBlocked Read/Write mechanism effectively pipelines the pager system. A simulation of the system at this moment, allows to verify that all processes run concurrently and synchronise when they want to

IPR2024-00476
Petitioner's Exhibit 1007
Page 000021

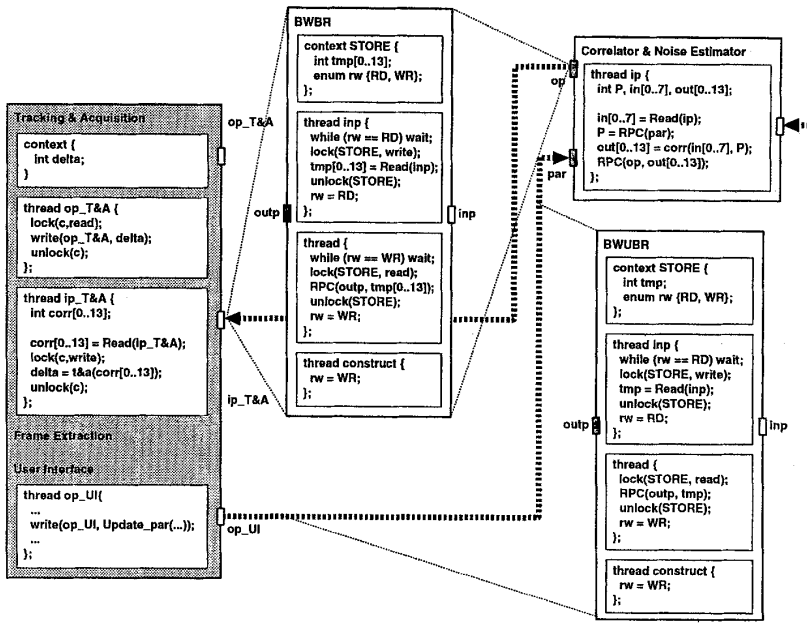


Figure 12. Introduction of Blocked Write–(Un)Blocked Read communication. The channel between ip_T&A and op is refined into the hierarchical BWBR channel. The channel between op_UI and par is refined into the hierarchical BWUBR channel.

perform I/O. Due to the introduction of the BWBR and BWUBR processes from Figure 12 the resulting C executable now contains a number of concurrent threads. In order to execute these threads concurrently, we use a multi-threaded C library [15].

Distributed simulation of the system is now[9] also an option. This requires that the user identifies which C processes to run on which work station. Per work station all the C processes are merged. Since the merged processes communicate with each other, the executables that eventually run on the work stations must also be able to communicate. This can be achieved via UNIX Inter-Process Communication between the executables on the work stations. To realise this in CoWare, the I/O ports of the merged processes are made hierarchical. Each hierarchical port, contains C code to communicate with a hierarchical port of another process. Once all the I/O ports are made hierarchical, these ports are merged with the previously merged processes. Each merged C process can then be compiled, linked with the multi-threaded C library and executed on the work stations.

IPR2024-00476
Petitioner's Exhibit 1007
Page 000022

3.4. Implementation of the Pager

After the newly introduced communication mechanism has been verified by simulation, each process has to be synthesised on its assigned target processor. In section 3.4.1 we discuss the implementation of processes in hardware. In section 3.4.2 we study the implementation of processes in software. In both cases the implementation involves a compilation and an encapsulation step. The compilation step consists of the actual implementation of the process in hardware or software. For this step we use existing compilers that generate processors with specific characteristics. These processors do not necessarily have a behavioural interface that is consistent with the original specification. Therefore an encapsulation step is required to encapsulate the generated processors in such a way that they are consistent with the original specifications. For processes that are implemented in hardware the encapsulation is rather straightforward. For processes that are implemented in software, such as the Tracking & Acquisition, the encapsulation is more complex. At the software side, it requires the introduction of I/O device drivers to link the software process to the physical processor on which it will run. At the hardware side, it requires the introduction of interfaces to link the physical processor to the hardware processors it has to communicate with.

3.4.1. Implementation of a Process in Hardware

Figure 13 illustrates the pure hardware implementation for the Correlator & Noise Estimator process and the merged Phase Correction and Chip Matched Filter process. This hardware implementation for the pager consists of three distinct steps:

- The (merged) DFL processes are synthesised by the CATHEDRAL silicon compiler. The compiler only generates processors of which all the inputs and outputs are of the master type. These processors are shown in Figure 13 as the inner rectangles.

- Each processor is encapsulated to make it consistent with the specification in which the DFL processes have slave inputs. In addition, the encapsulation includes clock gating circuitry to control the activity of the processor. The encapsulated processors are shown in Figure 13 as the big rectangles: they include the processor generated by CATHEDRAL and some encapsulation hardware. As can be observed the input ports of the encapsulated processors are now of the slave type. The encapsulation hardware is shown in detail in Figure 14.

- The BWBR process is implemented in hardware. In this case we obtain the gate-level implementation of this process from the library. This implementation is functionally equivalent to the original C-like description of these blocks in Figure 12. Figure 14 shows a detail of the implementation of the BWBR process that is used in the main signal path of the pager.

IPR2024-00476
Petitioner's Exhibit 1007
Page 000023

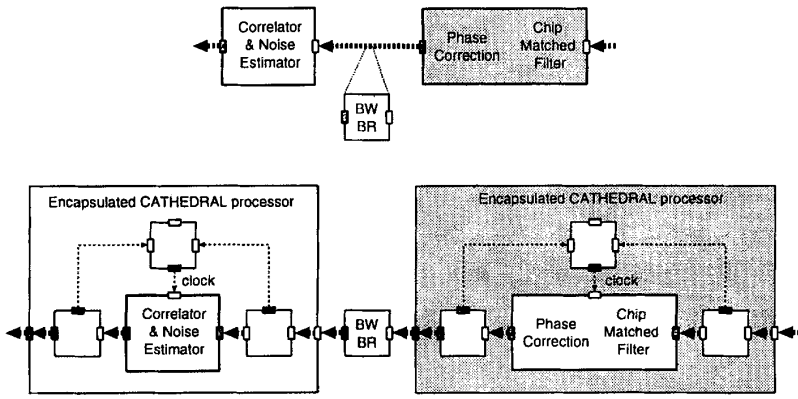


Figure 13. Implementation in hardware of the Correlation & Noise Estimator process and the merged Phase Correction and Chip Matched Filter process.

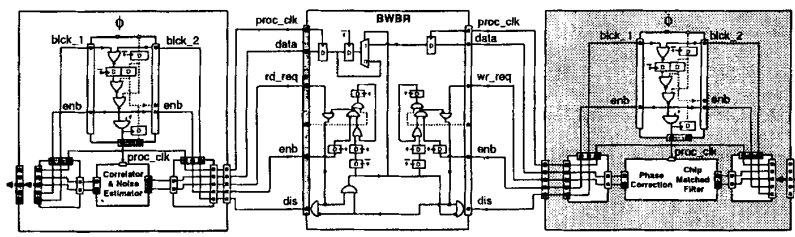


Figure 14. A typical communication buffer (CB) between two encapsulated accelerators. ϕ is the system clock.

3.4.2. *Implementation of a Process in Software*

Figure 15 shows the implementation in software of the Tracking & Acquisition, Frame Extraction and User Interface processes. In the figure we recognise the original merged C processes at the left-hand side, followed by a software I/O device driver, the software/hardware interface of the processor, a hardware interface, and finally the hardware processor that implements the Correlator & Noise Estimator process, preceded by the BWBR channel. The BWUBR communication channel between the User Interface process and the Correlator & Noise Estimator process is not shown in Figure 15. The main tasks to realise this implementation are the generation of the software I/O device drivers and the hardware interface to the ARM processor.

Software I/O Device Drivers. In the specification, the transfer of the 14 correlation values to the input port of the Tracking & Acquisition process is done via a single RPC. To

IPR2024-00476
Petitioner's Exhibit 1007
Page 000024

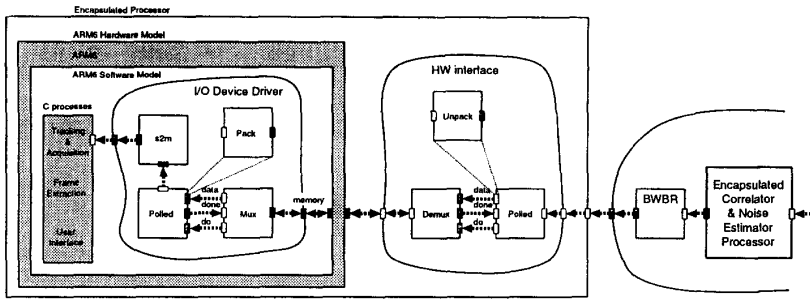


Figure 15. Implementation in software of the Tracking & Acquisition, Frame Extraction and User Interface processes.

implement this transfer, we have to choose one or more of the physical ports of the ARM. None of these ports is large enough to transfer the 14 correlation values simultaneously, so the transfer will have to be serialised. Also, since an RPC is not available as a primitive operation on the ARM, we have to implement an RPC in terms of ARM primitives on the physical ports of the ARM. The realisation of this software I/O device driver is done by SHOCK (see section 2.5) and consists of several steps.

- To transfer the correlation values a memory mapped I/O scheme is used via the memory port of the ARM. This port consists of a 32-bit address bus, a 32-bit data bus (which coincides with the 32-bit word length of each of the 14 correlation values) and a number of control terminals.
- Implementing the RPC in terms of ARM primitives on the memory port is done by the processes s2m, Polled and Mux. These processes are generic and are available from a library.
  - s2m. This process issues RPC calls to the Tracking & Acquisition process, transmitting data that it first retrieved from the Polled process (also via an RPC). The s2m process is the main thread in software. It initiates all other threads.
  - Polled. This process implements RPC in terms of ARM primitives by polling the memory port to see if there is data available. It does that by making use of three ports labelled do, done, and data. The process will wait until there are new data available. This is implemented by observing the do port. It will then retrieve that data (over the data port) and send it to the s2m process. Finally it will signal that it has received the data via the done port.
  - Mux. The ports do, done and data have to communicate with hardware that is external to the ARM via the memory port. Therefore, each of the ports, is mapped to a specific address. That address mapping is performed by the Mux process.
- The RPC from the Polled process to the Tracking & Acquisition process transmits all 14 correlation values in a single array via one RPC call. However, it is clear that we

IPR2024-00476
Petitioner's Exhibit 1007
Page 000025

cannot send these 14 correlation values over the memory port of the ARM as a single array. Hence, we need to refine the data master output port of the Polled process to retrieve the 14 correlation values one by one over the memory port and pack them into an array that can be sent to the Tracking & Acquisition process. This is achieved by the Pack process in which the single RPC call on the data port (from the Polled) process is translated into 14 RPC calls.

- The memory port of the ARM's software model contains the functions that describe how an actual memory access with a specific address is done in terms of ARM primitives. It is the ARM processor (hardware) itself that translates this into low-level protocol actions on the data bus, address bus and control terminals of the memory port.

- The original merged C process (Tracking & Acquisition, Frame Extraction and User Interface) and the processes of the I/O device driver (s2m, Polled, Pack and Mux) are now merged and compiled with an ARM-specific C compiler. The result is a VHDL process (a ROM with the machine code for the C processes including the device drivers).

Hardware Interface to the ARM Processor. The final step is to generate the hardware part of the interface with INTEGRAL [9]. Figure 16 shows all the inputs provided to INTEGRAL: the protocol of the memory port, the protocol of the BWBR channel, and the algorithmic description (provided by SHOCK) of the hardware interface performing the inverse function of the software I/O device driver. Starting from that information INTEGRAL will then automatically generate the hardware interface as shown in Figure 15.

The generic implementation script of Figure 4 also contains a final linking step to perform protocol conversion between possibly incompatible protocols. In the pager design, all protocols are compatible: the protocol of the BW(U)BR channels is tuned to the protocol of the CATHEDRAL processors to which they are connected and the protocol conversion for the ARM processor is taken care of by INTEGRAL during the hardware interface synthesis.

The single chip implementation of the pager [12] contains 340 K transistors occupying 47 mm^2 in a .6 μ technology and operates at a clock frequency of 40 Mhz.

4. Conclusions

Specification of complex systems requires various paradigms: SDF, DDF, hierarchical co-operating FSMs, etc. Specification languages, their simulators and synthesis tools are strongly linked to these paradigms. Hence the need for a *specification environment based on a data-model that allows the encapsulation of different host languages, simulation and synthesis paradigms.*

Complex systems are also heterogeneous at the architectural level. The implementation of a complex system consists of DSP processors, micro controllers, hardware accelerators, communication blocks, memory, etc.

The system design process must bridge the gap between the heterogeneous functional specification and the heterogeneous implementation of a complex system.

IPR2024-00476
Petitioner's Exhibit 1007
Page 000026

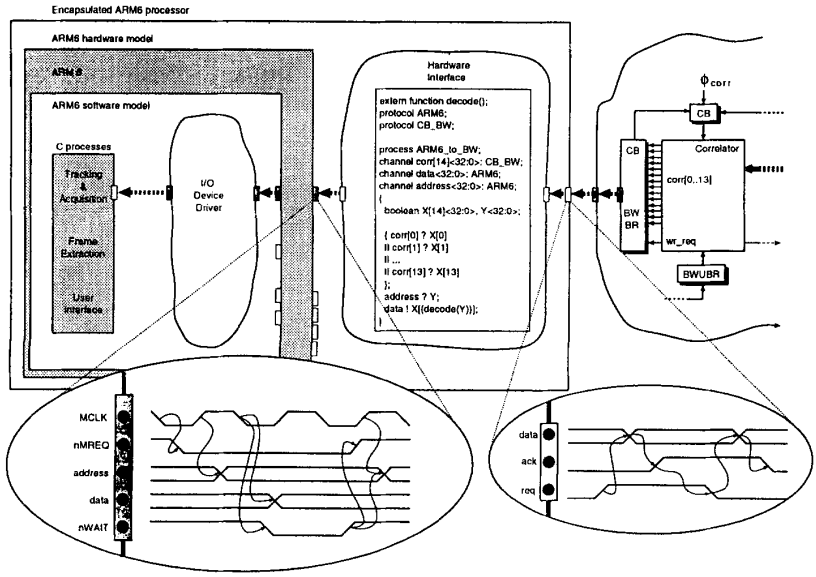


Figure 16. Interface synthesis between an ARM processor and the hardware correlator block. Given the protocol of the memory port, the protocol of the BWBR process, and an algorithmic description of the channel, INTEGRAL generates the implementation of the hardware interface.

Today's component compilers allow to synthesise 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 synchronisation. The allocation and binding step implies the hardware/software partitioning step.

The generation of the necessary hardware and software to make processors, accelerators, etc. and the environment communicate is the next problem. This involves

- Synthesis of communication hardware given the communication protocols and the behaviour of the channel.
- Synthesis of hardware and software for implementation of channels that connect software on programmable processors with hardware processors.
- Encapsulation of components so that they communicate with their neighbour components as specified in the functional specification.

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 allows for encapsulation of existing specifica-

IPR2024-00476
Petitioner's Exhibit 1007
Page 000027

tion languages, their simulators and design environments and forms the basis for system simulation and implementation.

The CoWare data-model is based on the concept of communicating processes and makes a strict separation between functional behaviour and communication. Processes can be described in their own host language to support specification heterogeneity. Host languages and their simulators can be reused through the concept of encapsulation. This allows for co-simulation and emulation on a UNIX work station. Constructs are provided to assign processes to components and existing component compilers can be reused through the concept of encapsulation. Finally, constructs are provided to describe and refine inter-process communication channels based on protocol libraries and component compiler encapsulation.

The design process for a spread-spectrum based pager system was described to show the feasibility of the CoWare data-model and the design flow implemented in the CoWare environment.

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 behaviour of complete complex systems is a problem.

Simulation as described in this contribution allows to verify the functional behaviour 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 behaviour 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 behaviour 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* behaviour of the system can be drawn.

When the communication mechanism of a complex system is refined, for example by pipelining the main DSP path in the pager design, the timing behaviour of the system may be radically altered. This might introduce deadlock or starvation of processes and hence analysis of the timing behaviour 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 [4] and code transformation verification [14], 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 complex system.

IPR2024-00476
Petitioner's Exhibit 1007
Page 000028

Acknowledgements

The research reported in this contribution was partly sponsored by ESA under the SCADES-2 and SCADES-3 projects.

Obviously much of the work would have been impossible without the cooperation of many colleagues. The work on CoWare is supported by A. Demarée, S. Samel, E. Umans, B. Vanthournout, and S. Vercauteren. The UNIX IPC communication was implemented by S. De Troch and S. Samel. The pager chip was designed by J. Vanhoof, V. Deruddere, S. Vernalde, M. Wouters, L. Philips, X. Gao, B. Vanhoof, and B. Gyselinckx who was especially helpful and patient in explaining the details of the design.

Notes

1. For conciseness we usually write VHDL, DFL and C process instead of the more accurate "primitive CoWare process described in VHDL, DFL and C".
2. Please note that in this context "VHDL process" means a primitive CoWare process described in VHDL. This VHDL code can be at the behavioural level or at the structural level. Hence, "VHDL process" does not denote a process in VHDL terminology (see also footnote 1 on page 361).
3. To construct a description of the final implementation, a black box view of the processor is sufficient. Simulation of the implementation is in that case not possible.
4. The language used in Figure 7 is pseudo-code meant to illustrate the RPC concept; it does not correspond to any of the languages mentioned in this document.
5. 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.
6. The sample rate is 4 complex samples/chip. The input rate for the Chip Matched Filter process is 8 complex samples/chip but it processes even and odd samples concurrently which leaves 2 cycles available for the processing of each sample.
7. The sample rate is 1 complex sample/chip because the Correlator & Noise Estimator process selects only one of the 4 complex samples/chip at its input.
8. This implies the availability of a description in C for the Correlator & Noise Estimator process (possibly derived from the DFL description with DFL2C).
9. Distributed simulation makes no sense for the specification. Since all communication is RPC based, only one process is active at the same instant anyway.

References

1. J. Buck et al., "The token flow model." in *Proceedings of the Data Flow Workshop*, Hamilton Island, Australia, May 1992.
2. J. Buck et al., "PTOLEMY: A framework for simulating and prototyping heterogeneous systems." *International Journal on Computer Simulation*, January 1994.
3. A. Fauth, J. Van Praet, and M. Freericks, "Describing instruction set processors using nML," in *Proceedings of the European Design and Test Conference, ED&TC 1995*, Paris, France, March 1995, pp. 505–507.
4. M. Genoe, L. Claesen, E. Verlind, F. Proesmans, and H. De Man, "Illustration of the SFG-Tracing multi level behavioral verification methodology," in *Proceedings of the ICCD 91*, Cambridge, MA, October 1991, pp. 338–341.
5. D.'Harel, "A visual formalism for complex systems." *Science of Computer Programming* (8), pp. 231–274, 1987.

IPR2024-00476
Petitioner's Exhibit 1007
Page 000029

6. P. N. Hilfinger, J. Rabaey, D. Genin, C. Scheers, and H. De Man, "DSP specification using the SILAGE language," in *Proceedings International Conference on Acoustics, Speech and Signal Processing*, Albuquerque, NM, April 1990, pp. 1057–1060.
7. D. Lanneer, J. Van Praet, K. Schoofs, W. Geurts, A. 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., Kluwer Academic Publishers, Boston, 1995.
8. E. A. Lee and D. G. Messerschmitt, "Synchronous data flow." *IEEE Proceedings*, September 1987.
9. B. Lin and S. Vercauteren, "Synthesis of concurrent system interface modules with automatic protocol conversion generation," in *Proceedings of the IEEE International Conference on Computer-Aided Design, ICCAD 94*, San José, CA, November 1994, pp. 101–108.
10. S. Narayan, F. Vahid, and D. D. Gajski, "System specification with the SpecCharts language." *IEEE Design & Test of Computers* pp. 6–13, December 1992.
11. S. Note, W. Geurts, F. Catthoor, and H. De Man, "Cathedral III: Architecture driven high-level synthesis for high throughput DSP applications," in *Proceedings of the 28th ACM/IEEE Design Automation Conference, DAC 91*, San Francisco, CA, June 1991, pp. 597–602.
12. L. Philips, J. Vanhoof, M. Wouters, B. Gyselinckx, I. Bolsens, and H. De Man, "A programmable spread spectrum modem for wireless communications," in *1996 On-Chip System Design Conference, Design Supercon 96*, Santa Clara, USA, Jan. 30–Feb. 1 1996, pp. 10.1–10.18.
13. J. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, "Fast prototyping of datapath-intensive architectures." *IEEE Design & Test of Computers* 8(6), pp. 40–51, June 1991.
14. H. Samsom, F. Franssen, F. Catthoor, and H. De Man, "Verification of loop transformations for real time signal processing applications," in *VLSI Signal Processing VII*, J. Rabaey, P. Chau, and J. Eldon, eds., IEEE Press, New York, NY, 1994, pp. 208–217.
15. UNIX System Laboratories, Inc. *USL C++ Language System Release 3.0 Library Manual*, 1992.
16. M. Van Canneyt, "Specification, simulation and implementation of a GSM speech codec with DSP station." *DSP & Multimedia Technology* 3(5), pp. 6–15, May 1994.
17. J. van Meerbergen et al., "PHIDEO: HIgh-level synthesis of high throughput applications." *Journal of VLSI Signal Processing* 9(1–2), pp. 89–104, January 1995.
18. P. Willekens et al., "Algorithm specification in DSP station using data flow language." *DSP Applications* 3(1), pp. 8–16, January 1994.

IPR2024-00476
Petitioner's Exhibit 1007
Page 000030