Interface Generation for Concurrent Processes During Hardware/Software Co-synthesis

Cristiano C. de Araújo and Edna Barros
Centro de Informática - UFPE
P.O. Box 7851 – Cidade Universitária
Recife - Brazil
cca2,ensb@di.ufpe.br

Abstract

This paper describes a model for interface, which is being used in the PISH co-design system. This model is based on layers and tries to keep the interface generation as independent as possible of the underlying target architecture. The proposed interface structuring in three layers provides abstraction of the communication implementation at process level and makes easier the interface generation process.

1. Introduction

With the growing complexity of the digital systems and the need for reducing the time to market, techniques for supporting hardware/software co-design have been developed in order to permit the joint specification, design and synthesis of mixed hardware/software systems [5][12]. Such systems consist of common-off-the-shelf (COTS) and ASIC components and have a variety of implementation technologies and interfaces, and a wide range of real-time data rates. The need for early prototypes to validate the specification and to provide the customer with feedback during the design process is another key factor motivating hardware/software co-design.

Some tools and methodologies supporting hardware/software co-design have been published in the last years [5] [6] [7][8][9][10] [12]. In most of them, however, once the initial description was partitioned, the interface between the hardware and the software components is synthesized by hand or in a semi-automated way.

This work takes into account the PISH co-design system, which allows the partitioning of occam descriptions by considering hardware/ software trade-off but also distinct hardware implementations [11]. Additionally, the correctness of the partitioning process can be assured through the use of formal verification techniques, in a constructive way [12][13] and a virtual prototype can be obtained in an early phase of the design process. The partitioning output is a set of communicating processes, some of them to be implemented in hardware, others in software and others for communication purposes. The next step is the generation of a real prototype, a very time consuming and error prone activity, and in the PISH co-design system it has been done by hand.

The complexity of the interface generation depends on the flexibility of the target architecture. Most systems with automatic partitioning taken into account a pre-defined target architecture, which makes the interface generation easier. But also in this case, automatic interface generation is not easy due to the semantic gap between the descriptions of the virtual and the real prototypes. Due to this fact, techniques for automatic interface generation is a feature of a small number of co-design systems [1][2][17].

For a correct interface generation two points should be considered; the possibility of communication generation among processes during the partitioning in a correct way and the availability of a method for automatic interface generation, which should be able to generate the software and hardware implementing the interface.

When the communication among modules is made explicit and assured to be correct, the mapping of the virtual prototype into the real one can be done in a more natural manner. Some approaches allows automatic communication generation [13].

The automatic interface generation is not a trivial task. Due to the dependence on the underlying target architecture, most approaches allow the interface generation for a fixed target architecture [1][10] or are specific for a domain application [3]. The main goal of this work is the development of an interface model for synchronous communication, which makes easier the interface generation process. The proposed model is based on layers, in order to allow communication actions at process level independent on the used processor. This paper is organized as follows: in the next section are the related works, section 2 gives an overview of the PISH co-design system including interface generation. A more detailed description of the proposed model is given in section 4. Section 5 illustrates an example. Some conclusions are presented in section 6.

2. Related Works

The interface in the POLIS approach [14] implements a domain specific communication mechanism between a set of co-design finite state machines (CFSM's). This mechanism uses asynchronous communication and is based

Proceedings of the 15 th Symposium on Integrated Circuits and Systems Design (SBCCI'02)
0-7695-1807-9/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

on event detection. The main problem with this approach is that it uses different approaches and protocol for the three different interfaces: hardware/hardware, hardware/software and software/software.

In [3] interfaces for synchronous dataflow (SDF) are automatically generated. This methodology uses a hierarchical approach to interface generation. A layered representation of the interface is given where in the first layer the communication is represented by abstract unidirectional links connecting source and sink nodes. These links are virtual channels. A second layer, implementation layer, is composed of computing elements (processors, microcontrollers, FPGA's), buses and memories. The automatic interface generation is performed by mapping the objects of the abstract layer to the implementation layer using the HASIS tool. This mapping is done by code generation not using component libraries.

In [2] an intermediate abstract architecture is also used. This architecture is composed of processing elements and point-to-point unidirectional channels. The processing elements can be hardware components or processors. In the latter case a hardware *wrapper* that encapsulates the software component as a hardware one is added to the processor core. This hardware wrapper implements the communication interface to its eternal environment. Unlike the previous one the automatic interface generation is done by choosing the right hardware wrapper components stored in libraries.

Our approach uses a domain specific mechanism like POLIS, but using a layered interface model like the latter two methodologies. But unlike them our implementation architecture is layered. Another characteristic of this work is that our layered model is symmetric, using the same protocol for the three interface types. In this way the automatic interface problem is restricted to the lower implementation layer as will be described in the following sections.

3. Communication among Processes in the PISH Co-design System

The approach for automatic interface generation is being developed in the context of the PISH co-design system. The PISH design methodology is depicted in Figure 1.

A system is specified using the occam language, the main reason to use occam is that, being based on CSP [16] occam has a simple and a elegant semantics, given in terms of algebraic laws, which allows a series of algebraic transformations to be performed on the original specification while preserving the semantics of the specification.

This initial specification is partitioned in hardware and software components. The set of transformation rules is applied according to the results of a cost analysis based on clustering techniques [11]. The output of the partitioning is a set of concurrent processes, which communicates through processes generated only for this purpose. This feature is an important support for the interface generation, since the communication among processes has been made explicit and is correct. The interface generation depends on the target architecture taken into account, so specific device drivers must be generated at software side, as well as specific hardware to make transparent for the hardware side which processor is being used. The interface between hardware modules must also be synthesized.

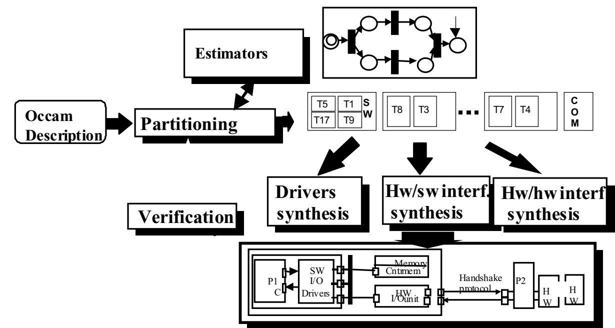

Figure 1- The PISH Design Flow

4. A layered Interface Model

In this section is explained the layered interface model used in the PISH co-design system. Table 1 shows communication abstraction levels as described in [19]. As one can see in this table the communication can be described in four abstraction levels from service level to register transfer level. At each level the communication behavior changes and more details are introduced.

At *service level* the communication behaves like a routing system where associations between processes are created. At this level the specification does not define timing or data type information. It is only worried on who communicates with who, how this is performed and what type of information is exchanged is treated by the other abstraction levels.

The *message level* is the second level in the hierarchy. In this case communication is performed through abstract communication channels connecting different processes. At this level no assumption is made about how the communication is going to be implemented, but contrary to the previous level, data type information is given. Here there is no protocol and processes in hardware or software communicate in the same way. The behavior of the communication is defined by high level constructs like occam communication mechanisms.

At *driver level,* primitive read and write functions are used to encapsulate the information in handshake protocols. Each one of the read and write operations hide the low level details of signal exchange during communication. At this

Proceedings of the 15 th Symposium on Integrated Circuits and Systems Design (SBCCI'02)
0-7695-1807-9/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

level communication time is predictable and is a function of the number of read/write operations performed during the execution of the communication protocol.

In the *register transfer level* communication is performed by specific signal exchanges. At this level bit signals are set or reset in a specific sequence of operations at every clock cycle. This level represents the physical implementation and the notion of time is given by the clock signal and set/reset delays. Processes are implemented as finite state machines that activate or deactivate the bit signals.

Abstraction Level	Communication Behavior
Service	Routing
Message	Protocol Conversion
Driver	Driver Level Protocol
Register Transfer	Transmission

Table 1 - Communication abstraction levels

4.1 PISH Implementation

When generating the interface of mixed systems composed of hardware and software components, the model taken into account should be able to represent the communication between hardware modules, the interaction among software processes, as well as the communication between hardware and software processes. When considering hardware/software communication, the generated interface is dependent on the used processor, on the communication protocol as well as on the communication media being used.

The high number of alternatives for implementing communication makes more difficult the interface generation in an automatic way. The main goal of this work is to develop an interface model, which allows decoupling as most as possible the interface parts of the architecture features. This has been achieved by organizing the interface into layers, as depicted in Figure 2

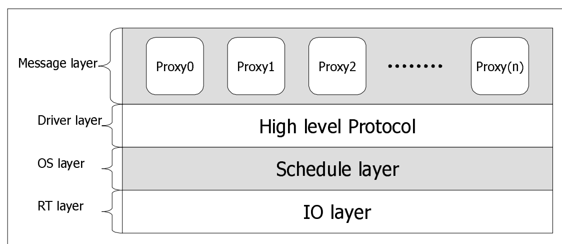

Figure 2 – The layers of Interface Model

Each layer includes software and hardware modules and makes transparent how the communication is done in the layer below. In the first layer, channels and drivers can be used for communication between hardware and software processes, as well as among hardware processes and among software processes. By using the modules on this layer, processes can communicate by using send/receive operations without know how the communication is implemented. This layer has a well-defined interface with the layers above and below and is independent on the used processor and on the used communication schema. The scheduling of all communication requests is also done at this layer. The data transfer itself is done at *io layer*, which is dependent on the used processor.

4.2 Proxy Layer

This layer represents the message level and is responsible for making the communication between processes using the interface transparent of each of the processes. As can be seen in Figure 2 (message layer) it is composed of proxies representing the communication channels. These proxies have the same interface as the channel, depending on how it is implemented (software or hardware). When one have two processes, one in hardware and the other in software, communicating through one channel, the hardware process sees the channel proxy in the interface as a hardware channel while the software process sees the proxy representing the same channel as a software channel. This way communication is made transparent for both hardware and software processes. Figure 3 shows the communication between processes *P0* and *P2* using the proxies *Px0* from the interface. For the process *P0* the proxy *Px0* represents a channel like *ch0* that it also uses for communication with *P1*. At this level all the implementation details of the communication are hidden from the processes and there is no difference between a process that uses a proxy or a channel.

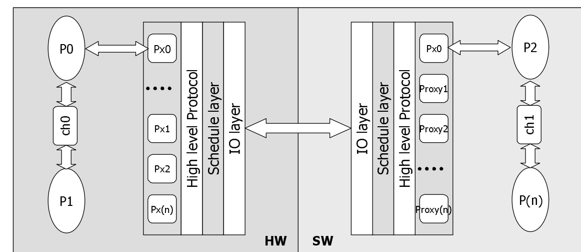

Figure 3 – Communication at message level

4.3 High Level Protocol Layer

This layer implements *Driver Level* in Table 1. Here are implemented the high level protocols used for transferring data from one processor to another one. These protocols are implemented by calling send/receive functions in hardware or software, depending on the processor that contains the interface. This layer brakes the high level data to be transferred in several concrete data types. By concrete data type one must understand as words that can be send

Proceedings of the 15 th Symposium on Integrated Circuits and Systems Design (SBCCI'02)
0-7695-1807-9/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

received over the processor's bus. Figure 4 illustrates the structure of a send driver implemented in software.

```
struct ChannelHS0 {
      T_flag send;
      T_flag ready;
      DataChannel0 dado;
};
```

a)

```
void sendChannelHS0 (DataType d, next_state, *state_var) {
   if (not channel activated) {
      put new data;
      activate channel;
   }
   else {
      if (transfer ready)
         go to next state
   }
}
```

b)

Figure 4 – A driver for send operation

4.4 Scheduling Layer

This layer is responsible for scheduling the access of the several concurrent proxies to the shared resource that is the hardware/software interface. As one see in Figure 3 there are several proxies in the message layer. Each one of these proxies can try to use the interface and this situation can occur in the same time. When this conflict happens the *scheduling layer* arbitrates and gives control access of the interface to just one of the proxies trying to use the interface. The *scheduling layer* guarantees a fair use of the interface by all the proxies that need to communicate. This way all the communication occurring in the system will be performed.

4.5 IO Layer

The IO layer performs the low level transfer of data from one processor to another one. As have been seen on 4.2 the high level protocols are implemented using the call of send/receive functions. How these functions are implemented is done in the *io layer.* A detailed view of the *io layer* is depicted in Figure 5.

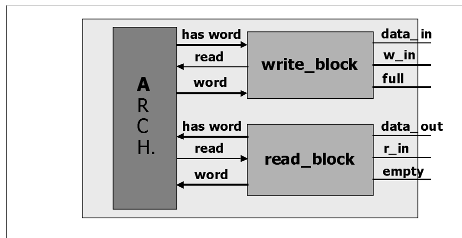

Figure 5– The IO Layer

It is composed of three blocks: architecture block, write block and read block. The *write block* is responsible for transferring data from the interface to the outside world, it checks whether there is a word to be send by the interface and if this is the case transfers it via the *architecture block.* The *read block* verifies whether there is a new word from outside the processor and receives it. These two blocks have the same structure independent on the underlying architecture. The *architecture block* interacts with the processor to see if data must be sent or received and performs the data transfer through the underlying interconnection media. For each new interconnection media a new *architecture block* must be designed and stored in a library for reuse in new projects.

5. A Method for Automatic Interface Generation

An overview of the methodology for automatic interface generation can be seen in Figure 6. Initially the occam description is partitioned in hardware, software and communication components. The result of this phase is another occam description that is guaranteed to have the same semantics as the original one. In a second step these components are translated to an internal format represented by Petri Nets. The Petri Net representation is then translated to C and VHDL for software and hardware implementation respectively. C code is generated for the software processes and communication and VHDL code for the hardware processes and communication components in hardware. Communication components in hardware and software represent the message, driver and OS layer of the interface model.

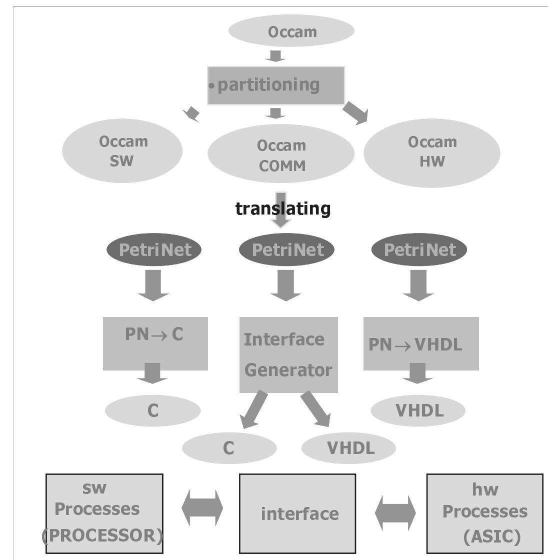

Figure 6 – A Methodology for automatic interface generation

The IO layer is implemented based on descriptions stored in a library. So the designer has to build its own library of low level interface elements. Despite the designer has to build this manually, it just do that once and can reuse the low level interface component in several projects.

IEEE COMPUTER SOCIETY

5.1 Automatic Interface Generation for an ATM Controller

In this section an ATM switch controller described in detail in [21] is used as an example for the interface generation methodology. The ATM switch controller must decide whether a cell must be sent or not based on four policy algorithms. The block diagram of the ATM switch is shown in Figure 7. Squared boxes represent processes and the round boxes represent communication channels. It is composed of the following processes: one routing table (TRGS), one cell reader, data reader block, four policy algorithms, one cell evaluator, cell sender and table update. The system is partitioned in hardware and software processes. Hardware processes are represented by the gray squared boxes while the software ones by the white squared boxes.

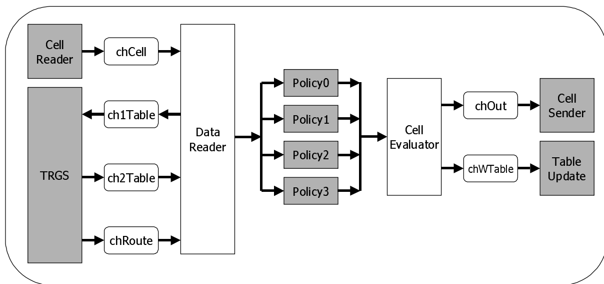

Figure 7 – ATM switch block diagram

The communication between the processes *cell reader* and *data reader* is detailed in Figure 8. Instead of the channel *chCell* the communication is performed through the the proxies *Px0* implemented in hardware and software. For the processes the proxies behave like the channel *chCell* and no change in the processes code is needed. The *high-level protocol layer* implements a simple protocol composed of the proxy id and a number of words that depends on the data size. When the proxies communicate they know the number of words involved in the transfer. The *schedule layer* is implemented as a round robin algorithm that always starts with the proxy following the last one that used the interface.

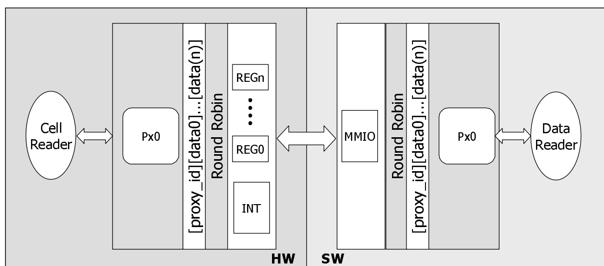

Figure 8 – Hw/Sw Process Communication

This guarantees that no proxy will wait indefinitely to access the interface. It should be noted thatse layer are the same in both sides of the interface. The last layer, *io layer,* is different, it has not the same implementation in hardware and software. In the target architecture the hardware processor is mapped as memory by the software processor. The scheme uses memory mapped io (MMIO). For transferring data from the hardware processor to the software one an interruption mechanism is used and reflected on the hardware *io layer* (INT), also in hardware are implemented a bank of registers that are read/written by the software processor. On the software side is implemented a MMIO mechanism for accessing the register on the hardware side.

6. Results

The tool extracts the concurrent threads from the Petri Net representation of the partitioned system. In this case 14 concurrent threads are generated and the results are summarized in the *Table 2*. The table gives the number of places and transitions for each thread and also its nature that can be hardware or software thread. The hardware threads are the 4 policy processes and can be noted thaty all have the same number of places and transitions. This comes from the fact that only the parameters are different, the policy processes are equal.

Thread type	Number
Software threads	10
Hardware threads	4

Table 2: Thread results

In *Table 3* the results for the IO threads selected by the system designer are shown. In this example all the IO threads are implemented in hardware. For each IO thread one VHDL file is generated.

Number of IO Threads	Type
6	Hardware

Table 3: IO Threads

As mentioned before the interface is implemented in layers. The last layer is responsible for implementing the 3 types of communication schemes between threads in hardware and software. One file is generated for each policy thread, resulting in four files. In table 4 are shown the results for the interface in hardware.

Block name	Type	Number of lines
Communication	Hardware	1089
Activation	Hardware	944
Finalisation	Hardware	932

Table 4: interface in hardware

The software implementation is simpler than the hardware one. In this case header and C files are generated

IEEE COMPUTER SOCIETY

for the parts of the system to be implemented in software. In *Table 5* are summarized the software results. The first file represents the whole system in software. It contains the main function. The second file, *processos.c*, implements the threads in software. The next file, *comunicacao.c*, implements the communication in software. As there are no IO threads to be implemented in software, no files are generated. The last three lines of the table contain the three layers of the interface. As in the case of the hardware interface, the *io_unit.c* file is generated based on a description of the target architecture while the others are generated automatically.

C file	Lines	H file	Lines
Atm_protocolo.c	58	-	-
processos.c	573	processos.h	11
comunicacao.c	1997	comunicacao.h	791
e_s.c	-	-	-
io_unit.c	40	io_unit.h	2
Comm_unit.c	230	comm_unit.h	17
prcs_unit.c	808	Prcs_unit.h	182

Table 5: software results

7. Conclusions

In this paper we have presented an interface model for synchronous communication, which is based on layers. The model includes three layers. The first layer, called the channel/drivers layer, consists of hardware and software channels and drivers to be used at process level. The second layer, the PRCS_unit controls the data transfer and scheduling of the communication through the interconnection media. The I/O_unit layer performs the data transfer between processor and the hardware part. In all these layers there are modules implemented in hardware and in software, which works in a complementary way. The layer organization of the interface and the similarity of modules belonging to the same layer make easier the automation of the interface generation process.

8. References

[1] Knudsen, P.V.; and Madsen, J., Communication Estimation for Hardware/Software Codesign, Proceedings of the International Workshop in Hardware/Sotware Co-Design - CODES 1998.

[2] B. Lin, S. Vercauteren, H. De Man*, Embedded Architecture Co-Synthesis and System Integration*, International Workshop on Hardware/Software Codesign, March 1996.

[3] M. Eisenring and J. Teich, *Domain-Specific Interface Generation from Dataflow Specifications,* Proceedings of the 6th International Workshop on Hardware/Software Codesign, March 1998.

[4] C. Araújo and E. Barros , Automatic Interface Generation among VHDL Processes in Hardware/Software Co-Design, FDL´99, August 1999.

[5] D. Gajski and F. Vahid, *Specification and Design of Embedded Hardware-Software Systems–*IEEE Design and Test of Computers, pp.53-67, Spring 1995

[6] T. BenIsmail, M. Abid, K. O´Brien and A. Jerraya,*„An Approach for Hardware/Software Codesign,* Proceedings of the RSP 94, França, 1994

[7] A. Kalavade , E. Lee, *A Hardware-Software Codesign Methodology for DSP Applications* – IEEE Design and Test of Computers, pp. 16-28, September 1993

[8] D. E. Thomas, J. K. Adams, H. Schmit, A Model *and Methodology for Hardware/Software Codesign–* IEEE Design and Test of Computers, pp. 6-15, September 1993

[9] R.K. Gupta. , C.N. Coelho , G. De Micheli, Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components– Proceedings of the 29th DAC, 1992

[10] R. Ernst , J. Henkel, T. Benner, *Hardware-Software Co-Synthesis for Microcontrollers–* IEEE Design and Test of Computers, pp. 64-75, December 1993

[11] E.Barros and W. Rosenstiel A Clustering Approach to Support Hardware/Software Partitioning". In: K. Buchenrieder, and J. Rozenblit (eds.), Computer Aided Software/Hardware Engineering. Chapter 11- IEEE Press, 1994.

[12] E. Barros and A. Sampaio,. *Towards Probably Correct Hardware/ Software Partitioning Using Occam*. In Proceedings of the Third International Workshop on Hardware/Software Codesign, (1994) 210-217, IEEE Press.

[13] L. Silva, A. Sampaio and E. Barros, A Normal Form Reduction Strategy for Hardware/Software Partitioning. In the Proceedings of the Conference Formal Methods Europe'97

[14] F. Balarin, A. Jurecska, and H. Hsieh et al. *Hardware-Software Co-Design of Embedded Systems : The Polis Approach.* Kluwer Academic Press, Boston, 1997.

[15] D. Pountain and D. May, *A Tutorial Introduction to OCCAM Programming.* Inmos BSP Professional Books, (1987).

[16] C. A. R. Hoare, *Communicating Sequential Processes* Prentice-Hall, 1985

[17] P. Chou, R.B. Ortega and G. Borriello, *The Chinook Hardware/Software Co-synthesis System*. Proceedings of the 8th International Symposium on System Synthesis. 1999.

[18] A. Baghdadi, D. Lyonnard, N-E. Zergainoh and A. Jerraya An Efficient Architecture Model for Systematic Design of Application-Specific Multiprocessor SoC. In Proceedings Design, Automation and Test in Europe, Los Alamitos, CA, 2001.

[19] K. Svarstad, G. Nicolescu and A. Jerraya A Model for Describing Communication between Aggregate Objects in the Specification and Design of Embedded Systems

[20] G. Nicolescu, S. Yoo and A. Jerraya Mixed-Level Cosimulation for Fine Gradual Refinement of Communication in SoC Design

[21] J. Yioda *ParTS – Uma Ferramenta de Suporte ao Particionamento Hardware/Software*. Recife: Universidade Federal de Pernambuco, 1999. Dissertação Mestrado.

0-7695-1807-9/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY