Domain-Specific Interface Generation from Dataflow Specifications

Michael Eisenring, Jürgen Teich
Computer Engineering and Communication Networks Lab (TIK)
Swiss Federal Institute of Technology (ETH), CH-8092 Zurich, Switzerland
WWW: http://www.tik.ee.ethz.ch/~{eisenri,teich}
email: {eisenring,teich}@tik.ee.ethz.ch

Abstract

In this paper, the problem of automatically mapping large-grain dataflow programs onto heterogeneous hardware/software architectures is treated. Starting with a given hardware/software partition, interfaces are automatically inserted into the specification to account for communication, in particular across hardware/software boundaries. Depending on the target architecture, the interfaces are refined according to given communication constraints (bus protocols, memory mapping, interrupts, DMA, etc.). An object-oriented approach is presented that enables an easy migration (retargeting) of typical communication primitives to other target architectures.

1 Introduction

This paper deals with the problem of automatic generation of hardware/software interfaces for certain classes of dataflow graph based specifications.

The variety of different *abstract communication styles* (e.g., buffered versus non-buffered, blocking versus non-blocking, synchronous versus asynchronous communication, etc.) encountered in different *dataflow process network* models [11], and typical *physical communication styles* such as via memory-mapped I/O, dedicated ports, etc., make it hard and inefficient to store all combinations of communication types (e.g., a channel implementing a blocking read, non-blocking write, buffered FIFO organization, reader in software, writer in hardware, etc.) in a library, possibly for all combinations and for each different target. Therefore, automatic interface generation tools are necessary which, at the same time, should be easily changeable (or reconfigurable) for other targets.

1.1 Motivation

Block-oriented schematic diagrams with dataflow semantics are widely used for describing digital signal processing applications (see, e.g., Fig. 1a). Examples of design systems that enable the specification of such systems are Ptolemy from UC Berkeley [3] and Cossap from Synopsys, to name a few.

In dataflow specifications, nodes represent computations, and directed edges between nodes represent the transfer of data between computations. A computation is deemed ready for execution whenever it has sufficient data on each of its input arcs. When a computation is executed, or *fired*, the corresponding node in the dataflow graph consumes some number of data values (*tokens*) from the input arcs and produces some number of tokens on the output arcs.

Example 1 *Figure 1a) shows the block diagram of a* molecular dynamics simulation *application. The method basically consists in iteratively computing forces on a set of molecules using Newton's equation of motion (block F) and updating the position of molecules (block Int) based on these computed forces. In order to reduce the (typically huge) number of expensive force computations, the list of pairs of particles for which nonimal forces exist, may be reduced by first eliminating those pairs for which the distance is greater than a given cutoff radius (Pair) and evaluating forces only for the remaining pairs. In order to reduce the computational effort of pairlist computation, the pairlist should only be updated every 5 iterations of the force-position loop.*

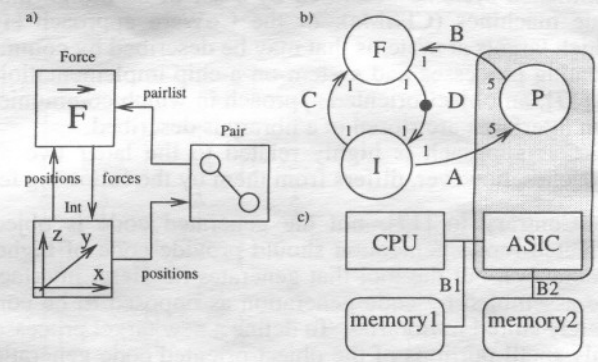

Figure 1. Dataflow graph for molecular dynamics simulation and target architecture

Dataflow models may be classified each having a different descriptive power such as SDF [12], BDF (boolean dataflow) [4], cyclo-static dataflow [7], and different dynamic dataflow models [9, 11]. In synchronous dataflow (SDF) [12], the number of tokens consumed from each input arc and produced on each output arc is a fixed positive integer that is known at compile time (see Fig. 1b).

Example 2 *In Fig. 1b), the schematic in Fig. 1a) has been refined to an SDF-graph: A node is enabled for firing once a fixed, statically known integer number of tokens (here 1 and 5) has accumulated on each input arc of an actor. Then,*

1092-6100/98 $10.00 © 1998 IEEE

the actor may fire: the node consumes these input tokens, performs its associated action (e.g., computing the set of forces in case of actor F), and transfers the result tokens to its outgoing arcs. Note that actor P (pairlist computation) is executed only every fifth iteration (multirate system).

The design systems named above may also generate code for simulation either in software (e.g., [2, 14]) or for prototyping an application in hardware [18]. Here, we seek to synthesize *mixed hardware/software solutions* from a specification of given particular dataflow network model. Such a synthesis path may later be exploited by a design space exploration tool (e.g., [16]). In this area, the automatic generation of interfaces (address decoders, interrupt circuitry, drivers, buses, etc.) has been addressed to a much less extent.

Example 3 *Figure 1c) shows a typical architecture for implementing the molecular dynamics simulation engine consisting of a microprocessor (CPU) and a hardware coprocessor (ASIC) as well as two memory blocks and two buses. The microprocessor implements the tasks of (irregular) force computations and position updates, the coprocessor implements the (regular) pairlist computations.*

The relevance of automatic compilation tools from high-level specifications onto hardware/software architectures has been recognized by many research groups, e.g. [8, 10, 15]. Here, we restrict ourselves to implementing dataflow models of computation, i.e., the *synchronous dataflow* (SDF) [12] model and focus on the generation of interfaces for hardware/software implementations.

Approaches to automatic interface generation include the CHINOOK [5] approach, as well as domain-specific approaches like in the Polis environment [1] for control-dominated systems described by a set of co-design finite state machines (CFSMs), or the CoWare approach [13] which targets at systems that may be described by communicating processes and system-on-a-chip implementations. In [17], an object-oriented approach in which communication interfaces are stored in a library is described.

Our approach is highly related to the latter two approaches, however, differs from them by the following features:

Contrary to [17], not the generated code is object-oriented (code generation should provide code of highest efficiency), but the tool that generates it. Here, interfaces are assembled by code generation as opposed to be completely stored in libraries. To define a new target processor, only small portions of the object-oriented code generation classes have to be rewritten. Contrary to CoWare [13], not only single-chip implementations are investigated. Also, the communication model is highly parameterized in that not only a particular scheme like Hoare's rendezvous may be implemented [13].

2 Methodology

We assume that a system is specified by a particular kind of *dataflow process network* [11], e.g., synchronous dataflow (SDF). The implementation process presented here is hierarchical. First, a layer given by *abstract models* used to describe dataflow graphs and target architectures is presented. The next section describes in detail the next lower layer of refinement presenting our *implementation models*.

2.1 Abstract models

2.1.1 Node

Figure 2a) shows the abstract model of a data processing node. An actor is parameterized by its inputs $\underline{i} = (i_0, i_1, \cdots, i_{n-1})$ and its outputs $\underline{o} = (o_0, o_1, \cdots, o_{m-1})$. The node functionality is given by a strict vector function $\underline{f}$. The semantics of a dataflow node may be described as follows [6] (see Fig. 2b).

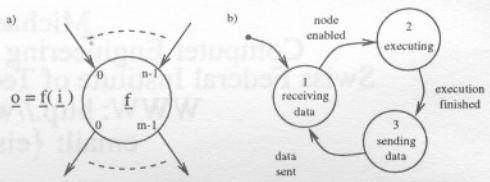

Figure 2. Abstract node model

receiving data: The node is waiting for data. If its firing rule is satisfied, it changes to state 2.

executing node function: The node function is performed exactly once. After the node has finished its operation, it changes to state 3.

sending data: The new tokens are written to the outputs. As long as not all of them have been communicated to its successors, it remains in this state. Finally, the node reenters state 1.

During hardware/software partitioning, a node is mapped to a computing resource, e.g., CPU, ASIC (see Fig. 1).

2.1.2 Edge

Edges represent directed communication between two nodes. The abstract view of an edge will be called *channel* (see Fig. 3).

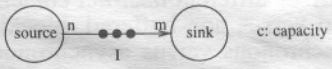

Figure 3. Abstract channel

A channel is a virtual unidirectional link between the source and sink node. Our abstract view makes the following assumptions:

Only *unidirectional* dataflow over the channel is allowed. Each channel has exactly one source and one sink node.

The channel may have a *local memory* with first-in first-out (FIFO) semantics for storing data. The FIFO may be initialized with individual data.

In case of SDF graphs, the abstract channel is characterized by six parameters: number of produced tokens n of each source node invocation, number of consumed tokens m on each sink node invocation, capacity constraint c, initial tokens I, *source* and *sink* node.

2.1.3 Target Architecture

We use a simple architecture model (see e.g., Fig. 4b). The model consists of three component types:

Computing resources (rectangular boxes): Each node in the dataflow graph is bound to a computing resource. Computing resources are of type *software* (microprocessor CPU, microcontroller MCU, DSP), and *hardware* (FPGA, ASIC).

Buses (solid lines): Each data transfer between computing resources and memories is bound to a bus resource (e.g., B1, B2, B3, B4 in Fig. 4b) which may be standard, uni- or bi-directional, or dedicated buses.

Memories (rounded boxes): Memories are used to store the channel FIFO data. Note that the binding possibilities of channel memory, source and sink nodes, and transfers are tightly dependent on each other (see also [16] for a graph model of these influences).

2.2 Implementation model

First, the refinement of a channel after the determination of the bindings is presented. During this refinement, all required parameters for final code generation of interfaces are defined. Then, an object oriented modeling of processors is discussed that eases the code generation for the target model.

A synthesis tool called HASIS (**ha**rdware **s**oftware **i**nterface **s**ynthesis) written in JAVA implements the described refinement steps and does the code generation for necessary interfaces.

2.2.1 Logical refinement of a channel

Our refinement approach is based on the introduction of *regular nodes* and *interface nodes* [6]. Regular nodes use *send* and *receive* primitives to communicate data along channels. Interface nodes are introduced if device drivers (in case of software) or hardware interfaces (in case of hardware) are necessary to be generated. The refinement process is described for a channel shown in Fig. 3.

Step 1: If a FIFO is needed (c>0) a new node (regular node) is inserted into the abstract channel specification (see Fig. 4a).

Step 2: Depending on the binding of regular nodes, additional interface nodes have to be inserted.

Example 4 *Fig. 4a) describes a binding of nodes and arcs of a given specification onto a target architecture (dotted lines). Now, an interface node I_{HW} has to be inserted that describes the interface to be synthesized for reading the FIFO data out of memory1 over bus B3 by the sink node implemented in hardware on the FPGA (see Fig. 4c). Depending on the target architecture, typically more than one interface node may have to be inserted and refined.*

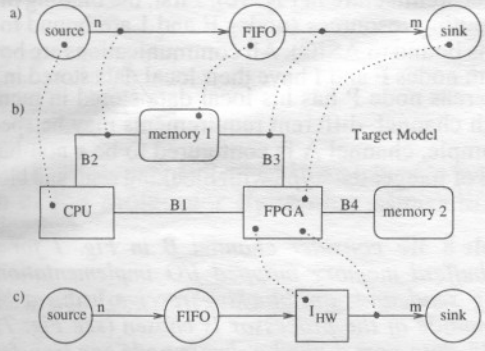

Figure 4. Logical channel refinement

Table 1 explains the parameters of each node. For example, source and sink node parameters include the function of the node but are independent of whether their input (output) is communicated using a blocking, non-blocking protocol, etc. The interface nodes (I_{HW} in this case) get information about the chosen communication style and target communication module.

	regular nodes	regular node	interf. nodes
parameter	**source sink**	**FIFO**	I_{HW} (I_{SW})
function	x		
binding	x	x	x
token datastructure	x	x	x
c		x	
n	source	x	x
m	sink	x	x
I		x	
(non-)blocking			x
synch./asynch.			x
communication module (2.2.3)			x

Table 1. Parameters of nodes for interface synthesis

2.2.2 Object oriented modeling of processors

In our implementation, each component of the target model (i.e., processor) typically possesses several I/O-signaling facilities, e.g., via interrupt, DMA, using dedicated I/O ports, etc. Figure 5 shows parts of a class tree for modelling these different facilites while exploiting the principles of classification, polymorphism, inheritance and composition. There

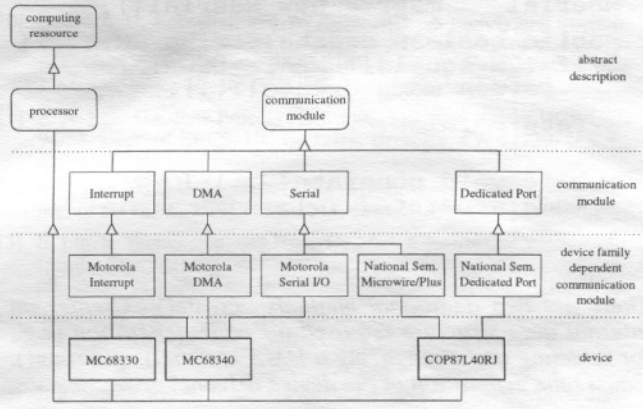

Figure 5. Object oriented modeling of processors

exist four class levels:

1. The abstract classes on the *abstract description* level describe common properties of the computing resources and the communication modules (e.g., each communication module has a generate(...) method for code generation of transfer commands).

2. There is one *communication module* class for each I/O-type (e.g., Interrupt, DMA, etc.) They define the very specific properties of each I/O block (e.g., which parameters specify a DMA module?).

3. The classes on the *device family dependent communication module* layer include the methods for code generation of I/O-primitives particular to the I/O-unit. They use parameterized templates which are stored in a device library.

4. On the *device* level, the processors are described as compositions of their communication modules.

Example 5 *The NATIONAL SEMICONDUCTOR COP87L40RJ microcontroller is a composition of the two modules* `NSC Microwire/Plus©` *and* `NSC Dedicated Port`.

This object oriented approach has the following major advantages:

1. *Reuse:* Each device-family-dependent communication module can be reused many times in order to model other processors of the same family.

Example 6 *The MOTOROLA processors MC68330 and MC68340 use the same communication module* `MOTOROLA Interrupt`.

2 *Ease of processor modeling:* The modeling of a processor is very easy by simply instantiating predefined communication modules.

Example 7 *The MOTOROLA processor MC68340 is modeled by a JAVA class, which instantiates the appropriate* device-family-dependent communication modules.

```
public class MC68340 extends Processor{
  MInterrupt mint = new MInterrupt();
  MDMA       mdma = new MDMA();
  MSerial    mser = new MSerial();
  public boolean create(String CM,...){
    if (CM.equals("DMA")==true)
      return mdma.create(...);
          :
    return false;
  }
  public void generate(...) {
    mdma.generate(...);
          :
  }
}
```

There are two additional methods: create(): *Creates an internal data structure required for code generation (e.g., for selecting and configuring a DMA channel).* generate(): *Causes the generation of the device drivers.*

3 *Retargeting and design space exploration:* Retargeting the interface generation is simply done by switching the processor in the target architecture and rerunning the code generation. Similarly, changes in the binding of nodes (e.g., during design space exploration) can be incorporated automaticaliy.

2.2.3 Code generation

The class `SystemSpec` (see Fig. 6) contains the specification of the dataflow graph, the architecture graph of the target, the binding of dataflow nodes to computing resources as well as a feasible schedule for the dataflow graph. The `CodeAssembler` manages the process of code synthesis:

1. *interface generation*: The `InterfaceGenerator` selects the appropriate interfaces for each channel and each participated processor (e.g., DMA channel: MC68340.create(DMA,...) in Example 7) and causes each processor instance to generate the interface code by using its communication modules (e.g., MC68340.generate(...) in Example 7).

2. *top level generation*: For each computing resource, the `CodeAssembler` invokes a generate method of the appropriate code generator (`CodeGeneratorC`, `CodeGeneratorVHDL`). They assemble the produced device drivers (C) and interface entities (VHDL) and generate a main program for processor resources and a top level VHDL-entity in case of dedicated hardware while considering a given scheduling mode (see [6] for scheduling).

3. *compilation*: The produced source code has to be compiled with appropriate compilers and synthesis tools.

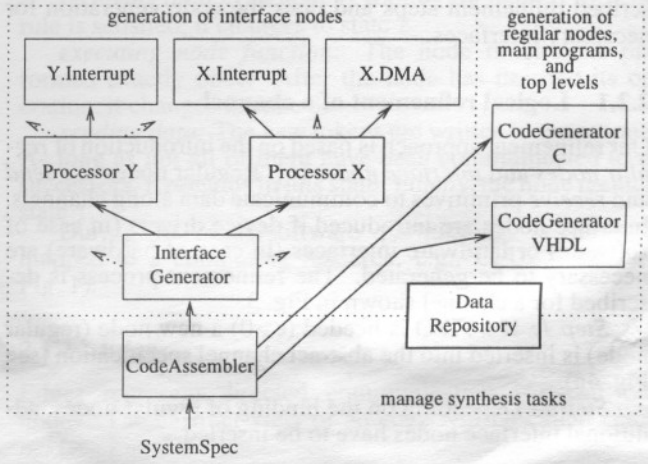

Figure 6. Detailed code generation

In all phases, the initial SystemSpec will be continously refined which is managed by the `DataRepository`.

3 Case study

As an example, we present the implementation of the molecular dynamics simulation application in Example 1 on the target architecture in Fig. 1c). First, the binding of nodes to computing resources (nodes F and I are bound to CPU, node P is bound to ASIC). All communications are bound to B1. Both nodes F and I have their local data stored in memory1 wereas node P has his local data stored in memory2. For each channel, different requirements may be specified. For example, channel A is configured to be a non-buffered channel of integer datatype with blocking send and blocking receive semantics.

Example 8 *We consider channel B in Fig. 1 for which a non-bufferd memory mapped I/O implementation with blocking send and non-blocking receive using a built-in DMA module of the processor is chosen (see Fig. 7). Via the DMA data port d, the hardware node can transfer data to the software node. An additional status port s consists of one simple data ready bit R used for communication synchronisation. The flowcharts describe the semantics that the*

interfaces implement. The non-blocking receive primitive of node F has the two local states stop *(initial state) and* run *with different behaviour to consider the DMA module status. The communication is done in two steps: (1) By setting the R bit, the hardware signals a demand for communication. (2) The data values are transmitted when the DMA module issues an explicit read access on the data port.*

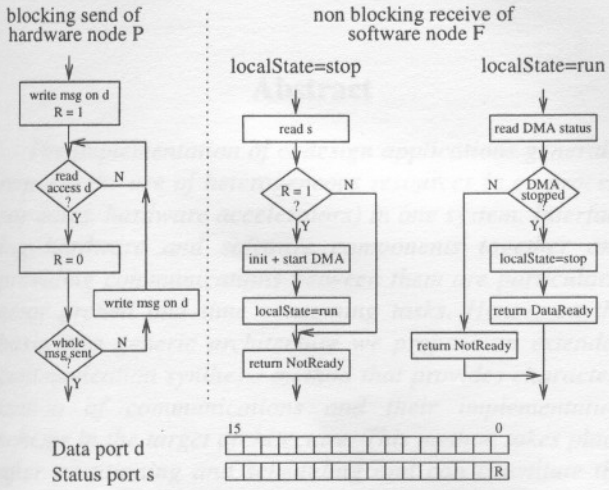

Figure 7. Interface refinement for channel B

Figure 8a) shows a code template for the software node F. a) The node behaviour is given by a C-function. b) Now, a node function is generated for each software node (e.g., F1 for F) using the communication primitives (e.g., recvB(),...) to read and send data. The semantics of the communication primitives depends on the actual channel specification. c) The complete software program consists of the two functions F1(), I1(), the generated communication primitives (recvB(),sendD(),recvD(),sendA(),sendC(), recvC()) and a scheduler. Additionally, there is a main() function in which the initialization of communication primitives and the initialization of memories with initial data takes place. Then, main() calls the scheduler. d) The scheduler is created by HASIS, too. In the example, static-scheduling [6] has been chosen (Fig. 8d).

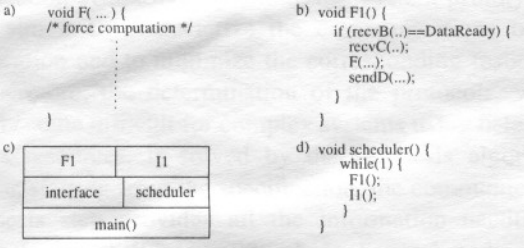

Figure 8. Implementation of the software partition of the molecular dynamics simulation application

4 Acknowledgement and further work

We would like to thank Martin Gerber and Lothar Thiele for discussions on the molecular dynamics example. Note that using a cyclo-static model would describe the communication in this application in a more efficient way. In the future, it will be shown how the concepts described here only for SDF extend to cyclo-static dataflow and more general dataflow models.

References

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

[2] S. S. Bhattacharyya and E. A. Lee. Scheduling synchronous data flow graphs for efficient looping. *Journal of VLSI Signal Processing*, 6:271–288, 1993.

[3] J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal on Computer Simulation*, 4:155–182, 1991.

[4] J. T. Buck. Scheduling dynamic dataflow graphs with bounded memory using the Token Flow Model. Technical Report UCB/ERL 93/69, Ph.D dissertation, Dept. of EECS, UC Berkeley, Berkeley, CA 94720, U.S.A., 1993.

[5] P. Chou, R. Ortega, and G. Borriello. Interface co-synthesis techniques for embedded systems. In *Proc. of the IEEE/ACM Int. Conf. on CAD (ICCAD)*, pages 280–287, San Jose, November 1995.

[6] M. Eisenring, J. Teich, and L. Thiele. Rapid prototyping of dataflow programs on hardware/software architectures. In *Proc. of HICSS-31, Proc. of the Hawai'i Int. Conf. on Syst. Sci.*, Kona, Hawaii, January 1998.

[7] M. Engels, G. Bilsen, R. Lauwereins, and J. Peperstraete. Cyclo-Static Data Flow: Model and implementation. In *Proc. 28th Asilomar Conf. on Signals, Systems, and Computers*, pages 503–507, Pacific Grove, CA, 1994.

[8] R. W. Hartenstein and J. Becker. A two-level co-design framework for data-driven Xputer-based Accelerators. In *Proc. of the 30th Annual Hawaii Int. Conf. on System Science (HICSS-30)*, Wailea, Hawaii, U.S.A., January 1996.

[9] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*, North Holland, 1974.

[10] G. Koch, U. Kebschul, and W. Rosenstiel. A prototyping environment for hardware/software codesign in the COBRA project. In *Proc. of Codes/CASHE'94 - the 3rd Int. Workshop on Hardware/Software Codesign*, pages 10–16, Grenoble, France, September 1994.

[11] E. A. Lee. Dataflow Process Networks. Technical Report UCB/ERL 94/53, Dept. of EECS, UC Berkeley, Berkeley, CA 94720, U.S.A., 1994.

[12] E.A. Lee and D.G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[13] B. Lin, S. Vercauteren, and Hugo De Man. Constructing application-specific heterogeneous embedded architectures for custom hw/sw applications. In *ACM/IEEE Design Automation Conference*, June 1996.

[14] S. Ritz, M. Pankert, and H. Meyr. High level software synthesis for signal processing systems. In *Proc. Int. Conf. on Application-Specific Array Processors*, pages 679–693, Berkeley, CA, 1992.

[15] J. Rozenbilt and K. Buchenrieder. *Codesign: Computer-Aided Software/Hardware Engineering*. IEEE Press, 1995.

[16] J. Teich, T. Blickle, and L. Thiele. An Evolutionary Approach to System-Level Synthesis. In *Proc. of Codes/CASHE'97 - the 5th Int. Workshop on Hardware/Software Codesign*, pages 167–171, Braunschweig, Germany, March 1997.

[17] F. Vahid and L. Tauro. An object-oriented communication library for hardware-software codesign. In *Proc. of Codes/CASHE'97 - the 5th Int. Workshop on Hardware/Software Codesign*, pages 81–86, Braunschweig, Germany, March 1997.

[18] P. W. Zepter. *Programmgestützter Entwurf integrierter Schaltungen für die digitale Nachrichtenübertragung aus Datenflussbeschreibungen.* PhD thesis, Lehrstuhl für Integrierte Systeme der Signalverarbeitung, TH Aachen, Germany, 1995.