Rapid Prototyping of Dataflow Programs on Hardware/Software Architectures

Michael Eisenring, Jürgen Teich, Lothar Thiele

Computer Engineering and Communication Networks Lab (TIK)
Swiss Federal Institute of Technology (ETH), CH-8092 Zurich, Switzerland
email: {eisenring,teich,thiele}@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 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.). A framework is described that uses an object-oriented approach to transform a given dataflow graph and to generate code for the actors as well as for the interfaces. The object-orientation enables an easy migration (retargeting) of typical communication primitives to other target architectures.

1. Introduction

With single chip solutions comprising millions of transistors today, the requirements in size and speed of systems, in areas such as embedded control and data processing applications have risen accordingly. The heterogeneity of such target systems requires the integration of knowledge from many different areas such as software compilation and hardware design. Whereas for both domains, efficient tools for synthesis are available today, tools for *simulation* and automatic *interface synthesis* are still missing or immature and turn out to become one of the major bottlenecks that prevent short design cycles in the design of hardware/software systems.

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

An analysis of *abstract communication types* (e.g., buffered versus non-buffered, blocking versus non-blocking, synchronous versus asynchronous communication, etc.) encountered in different *dataflow process network* models [14], and typical *physical communication types* such as memory-mapped I/O, interrupt or DMA-transfer, 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. Furthermore, these tools should be easily changeable (or reconfigurable) for other targets.

These challenges provide the motivation for the following work which is part of the CodeSign project at ETH Zurich.

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 [4] and Cossap from Synopsys, to name a few.

Dataflow [7] can be viewed as a graph-oriented programming paradigm in which 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. Dataflow imposes only partial ordering constraints, thus exposing parallelism.

Example 1 *Figure 1a) shows an application of a digital FIR-filter that receives a stream of input data from*

a source node and, after processing an input, outputs the result of the filter to a display (right hand side).

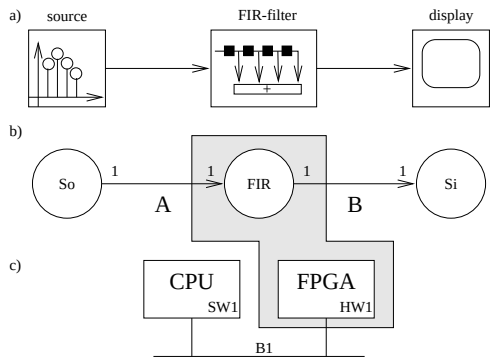

Figure 1: Dataflow graph and target architecture

Concerning the communication rules, different dataflow models may be classified each having a different descriptive power such as SDF [15, 16], BDF (boolean dataflow) [5], cyclostatic dataflow [2, 8], and different dynamic dataflow models [12, 14]. Hence, the blocks in a schematic correspond to dataflow nodes (frequently also called *actors*) and the wires correspond to arcs in a dataflow graph. In synchronous dataflow (SDF) [16], 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 the ones associated with the arcs in 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) has accumulated on each input arc of an actor. Then, the actor may fire: the node consumes these input tokens, performs its associated action (e.g., computing a FIR-filter output), and transfers the result tokens to the outgoing arcs.*

The design systems named above may also generate code for simulation in software (e.g., [20, 18, 1]) or for prototyping an application in hardware (e.g., by generating VHDL-code [23]).

However, a pure software solution or a pure hardware solution may not be the optimal choice for implementing a dataflow model. For example, a software solution may be too slow to satisfy real-time data-rate constraints. A pure hardware solution may be too expensive, and hence overdesigned.

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., [3, 21]) in order to explore optimal architectures.

Example 3 *Figure 1c) shows a typical architecture for implementing the FIR-filter on a mixed hardware/software platform including a microprocessor and a hardware coprocessor (FPGA) for hardware prototyping. The microprocessor implements non-time-critical tasks (source, display), the coprocessor implements the filter.*

Whereas there exist already tools for generating either hardware or software for certain dataflow process networks, their prototyping on mixed hardware/software platforms involves also the generation of interfaces, including hardware interfaces (address allocation, address decoding, interrupt circuitry, drivers, buses, etc.) and software interfaces (device drivers, interrupt service routines, etc.).

1.2. Existing Work

The importance and usefulness of automatic compilation tools for high-level specifications onto hardware/software architectures has been recognized by many research groups, e.g., [17, 13, 19, 9, 10]. Some approaches consider very broad application classes [13] or the mapping of complete languages like OCCAM [17]. Here, we restrict ourselves to implementing dataflow models of computation, i.e., the *synchronous dataflow* (SDF) [16] model and focus on the generation of hardware/software implementations by rapidly prototyping dataflow graphs.

The approach described by Bhattacharyya [1], e.g., treats the problem of optimally generating code for uniprocessor (all nodes implemented in software) implementations of SDF-graphs. The construction of multiprocessor schedules has also been attacked by several authors, e.g., [20]. However, not at the level of rapidly prototyping such solutions on heterogeneous architectures. The construction of mixed hardware/software schedules involving the generation of a partition of an SDF-graph into actors realized in hardware and actors for which a processor schedule is derived, has so far been considered to a much less extent.

A similar approach to map SDF-graphs and extensions thereof into hardware is described by Zepter [23]. However there, only pure hardware solutions are considered.

Here, we focus at mixed hardware/software solutions and describe a methodology of how to rapidly prototype dataflow computations on heterogeneous hardware/software architectures.

1.3. Highlights of the CodeSign approach

Although there exist already efforts for describing and summarizing the basic communication and interface

types to implement process network models (e.g., the object-oriented library-based approach described [22]), we believe that the combinatorial complexity of possible communication types and ways to implement them makes it impossible to store communication modules and interfaces completely in a library, possibly even in different libraries, one for each target architecture.

Therefore, the major focus of the work described in the following is to

- *characterize the essential properties of abstract communication types* typical for different dataflow models. This will result in the definition of a communication refinement called *channel*,
- *characterize the essential physical communication types* (e.g., memory-mapped I/O, interrupts, etc.) and extract their properties, and
- describe an *object-oriented code generation approach* that enables the designer to easily retarget the code generator for a channel (interface synthesis) to different heterogeneous target architectures. Note that not the generated code is object-oriented (code generation should provide code of highest efficiency), but the tool that generates it. Contrary to [22], interfaces are assembled by code generation as opposed to be stored in libraries. Hence, no complete libraries have to be written to define a new target, but only the portions of the code generation methods.

2. Methodology

As presented in the previous section, we assume that a system is specified by a particular *dataflow process network* [14], i.e., synchronous dataflow.

The implementation process presented here is hierarchical. First, a layer given by *abstract models* used to describe nodes, edges, and target architectures is presented. The next section describes the next lower layer of refinement presenting *implementation models*.

2.1. Abstract models

The initial specification of a system should not contain information or anticipate decisions about the final implementation style (software or hardware) and technology (FPGA, ASIC, etc.). Therefore, the basic attributes of each needed component will be identified by the following abstract models.

2.1.1. Node

The following constraints are to be considered:

- The same model of a dataflow actor should be used for nodes being refined to hardware or software. Therefore, its behaviour is specified purely functional at this level.
- The node has to be independent from the implementation technology of its communication partners.

The model in Fig. 2a) satisfies these constraints. 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 $\underline{f}$.

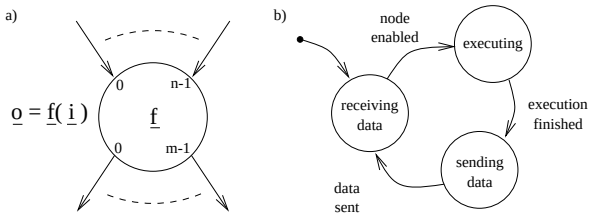

Figure 2: Abstract node model

The semantics of a dataflow node may be described as follows. The node is enabled for firing depending on the communication rule of the chosen dataflow model (e.g., if each input contains at least one token in case of the model of *marked graphs* [6]). It calculates the associated function and outputs the tokens to the output edges. We assume that a dataflow actor has three states (see Fig. 2b):

1. *receiving data*: This state represents the initial state of each node at startup time. The node is waiting for data. If its firing rule is satisfied, it changes to state 2.
2. *executing node function*: The node function is performed exactly once. After the node has finished its operation, it changes to state 3.
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.

This abstract model is common to several dataflow process network models and serves as the basis for later refinement in software and hardware.

2.1.2. Edge

An edge is assumed to be a dedicated, directed link between two nodes. An edges transports data which is produced by its source node and consumed by its sink node. The abstract view of an edge will be called *channel* (see Fig. 3).

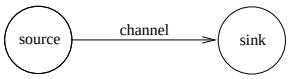

Figure 3: Abstract channel

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.
- The channel semantics is independent of a later refinement in hardware or software.

It turns out that the usage of libraries to store different types of channel implementations is not efficient because of the many possible communication types between components. Therefore, a code generation approach is taken, and a channel implementation is generated for each communication association. The channel specification includes information about the

- communication partners, which
- protocol to use (semantics of communication),
- type and number of data to communicate,
- FIFO size (and other memory attributes).

The process of channel synthesis creates a dedicated and optimized channel for each edge of the given dataflow graph.

2.1.3. Target Architecture

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

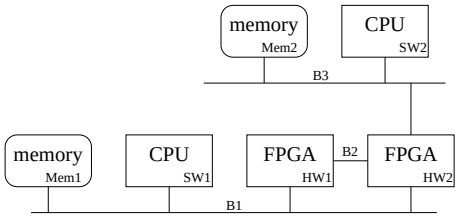

Figure 4: Target Model

- *Computing resources* (rectangular boxes): Each node in the dataflow graph is bound to a computing resource. Computing resources are of type *software* (microprocessor, microcontroller, DSP, e.g., SW1, SW2), and *hardware* (FPGA, ASIC, e.g., HW1, HW2).
- *Buses* (solid lines, e.g., B1, B2, B3). Each edge of the dataflow graph is bound to one bus.
- *Memories* (rounded box): Memory to store communicated data is introduced later in the refinement process (as part of FIFOs described in Section 2.2.1). If a channel is bound to a particular bus, then its memory can only be bound to a memory block connected to this bus.

2.2. Implementation models

The following *implementation models* refine the abstract models by adding information about the physical implementation. The complete refinement process consists of assembling together *basic components*.

2.2.1. Basic components

We consider the example of the refinement of a channel where the sending node is realized in software and the receiving node is realized in hardware (see Fig. 5). The

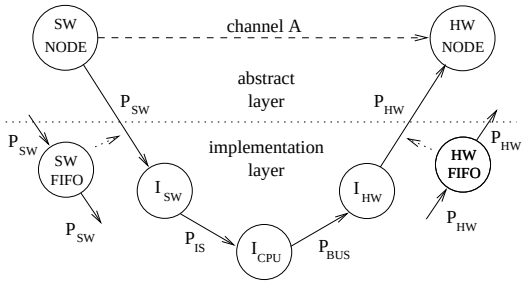

Figure 5: The hardware/software channel

elementary set of needed basic components can be partitioned into two types of nodes: *interface nodes* and *regular nodes*. The regular nodes are the nodes hardware node, software node, software FIFO, and hardware FIFO. The interface nodes I_{SW}, I_{HW} and I_{CPU} are used when regular nodes of different *protocols* are connected.

1. *Software buffer* $SWFIFO$: The software node in Fig. 5 issues a send communication. A software buffer may be inserted if necessary. Its structure will be described in Section 2.4.1.

2. *Interface* I_{SW} between software node and microprocessor CPU: This node represents the device driver to realize the software communication function (send) for the data transfer. It depends on the initial channel requirements (e.g., DMA).

3. The *microprocessor interface* I_{CPU}: The CPU determines the possible protocols P_{IS} which are commands for controlling peripheral devices and P_{BUS}. The latter represents protocols given for signals watchable at processor pins, e.g., bus transfer protocols.

4. The *hardware interface* I_{HW} (realized in hardware) translates the bus protocol P_{BUS} into the protocol used between hardware nodes. This is the hardware part of the channel and depends on the channel requirements as I_{SW} does.

5. *Hardware buffer* $HWFIFO$: If a buffer is desired in hardware, it will be inserted at this point. The structure is described in Section 2.4.2.

2.3. Protocols and interface nodes

The meaning of the interfaces, and the intermediate protocols are explained now in more detail:

1. P_{SW} stands for function calls at the high-level language level, e.g., send and recv commands. Here, these calls have the syntax

   `(send | recv)⟨ChannelName⟩(MessagePointer)`

   e.g., `sendA(msg)` or `recvA(msg)`. This protocol is used for communication along all input and output channels of software nodes.

2. P_{IS} represents the communication in terms of a sequence of assembly language commands given the instruction set of the CPU. This code sequence depends on the used communication mode, e.g., commands for interrupt handling, device driver initialization, etc.

3. P_{BUS} represents the protocols observable at the pins of the processor chip (cycle-level). It depends on the type of microprocessor and is specified in the datasheet of the CPU.

4. P_{HW}: In terms of hardware, there must exist a bus for communicating the data (databus) and a controlbus for synchronisation. In our environment, this bus called `HWBUS` consists of a databus and one or two handshake lines for synchronisation (see Fig. 6a).

Depending on the implementation style, different interface implementations for hardware (I_{HW}) and software (I_{SW}) are generated, for example for communication via interrupts, etc.

Although there seem to be a lot of different channel implementation possibilities for I_{HW}, they have all the basic structure shown in Fig. 6b). A microprocessor bus in general consists of the three components data-, control- and addressbus. If the microprocessor reads or writes on addresses allocated to the ASIC, the address decoder recognizes these requests and selects the appropriate memory location. The memory builds a simple data buffer between hardware and software. Synchronisaton is needed for proper exchange of data.

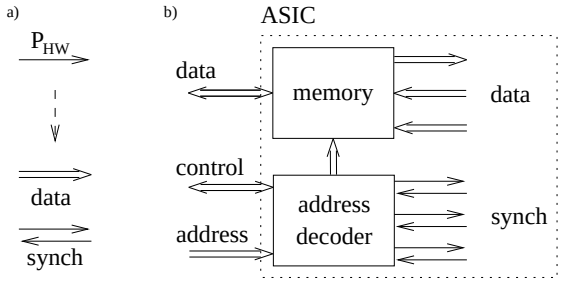

Figure 6: a) `HWBUS` b) Interface structure for I_{HW}

2.4. Regular nodes

2.4.1. FIFO in software

The structure of a channel memory with FIFO semantics that is realized in software is shown in Fig. 7.

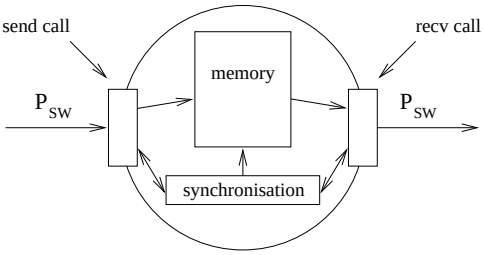

Figure 7: Fifo in software: $SWFIFO$

It basically consists of five different components: (1) a write access of the sending node (send call), (2) a read access of the receiving node (recv call), (3) the memory cells for data buffering, (4) an FSM for synchronisation that controls the access to the FIFO, and (5) a function for initializing data values in the memory.

The datatype and FIFO depth of a channel are taken from given channel parameters. The process

of channel synthesis allocates a corresponding memory segment and generates an appropriate synchronisation FSM and the read and write access functions.

2.4.2. FIFO in hardware

The structure of the hardware FIFO (see Fig. 8) is very similar to the software FIFO. It is named $HWFIFO$.

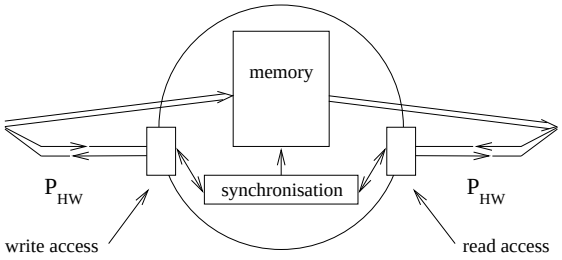

Figure 8: Fifo in hardware: $HWFIFO$

The physical datasize (buswidth of connected bus, dataformat) plays a very important attribute here. For example, FPGAs have only a limited number of routing resources. So sometimes, it is necessary to implement a datawidth smaller than the word size of data to be transfered along a channel. The synthesis tool splits the given word size into portions of the buswidth and generates an interface which transfers them sequentially. For the described model, synthesizable and retargetable VHDL-code [11] is generated.

2.4.3. Software node

The functionality of a software node is given in the form of a simple C-function (see e.g., the function SW_NODE in Fig. 9a). The formal parameters build the interface to the outer world. No communication primitives are used in this functional description. Therefore, the node functionality may be stored in a target independent library.

a)
```
void SW_NODE(int in, int *out) {
    static int LocalValue;
                  .
                  .
                  .
    *out = LocalValue;
}
```

b)
```
void SW_NODE1() {
    int rm, val1, val2;
    while(1) {
        rm = recvA(&val1);
        SW_NODE(val1, &val2);
        rm = sendB(val2);
    }
}
```

Figure 9: Implementation of a software node

Example 4 *Figure 9a) shows an example of a how a node function can be specified for a node with one input and one output.*

Now, this C-function is invoked in a second generated function (see the function SW_NODE1() in Example 5) that describes the read along input channels, the processing of the node, and the sending of data along output channels.

Example 5 *For communication with other nodes, the new function SW_NODE1 in Fig. 9b) is generated using the (also generated) communication primitives recvA() and sendB().*

2.4.4. Hardware node

Besides the requirements to implement a certain node functionality and communication protocols, there exist constraints with respect to compilation. Some of them are reported below: (1) Hardware compilers still require much higher compilation times than software compilers to implement the same functionality. (2) When using high-level synthesis tools, there are restrictions in codesize and therefore in complexity, too. Often, they may treat only a few hundred lines of code at most. (3) Only a small subset of VHDL or Verilog is supported for hardware synthesis.

These constraints led to a hierarchical code generation scheme, see Fig. 10. On the highest level a), the pure functional description of a node is given in the form of the VHDL entity HW_NODE.

Handshake lines are inserted for each channel into the primer functional specification (see Fig. 10b). They are used to implement the signaling of data availability and completion of data transfers. Two more signals are inserted (start, ready). The signal start initiates the computation, ready signals the end of the computation. This new behavioral description of the code is now compiled by the Synopsys behavioral compiler into a register transfer level (RTL) description. As shown in Fig. 10b), the internal structure of this description consists of a datapath, memory and the control FSM.

In a second step, the missing communication interfaces are added in the form of protocol FSMs, see Fig. 10c). The node is now fully assembled according to the initial requirements of functionality and communication protocol and may be used (instantiated).

2.5. Scheduling

It is assumed that the *binding* of actors to computing resources is done statically, i.e., at compile-time.

Another question deals with the problem of *scheduling* actors ready for execution. Here, it is assumed that all actors mapped to hardware resources are able to fire concurrently, if simultaneously enabled. Hence,

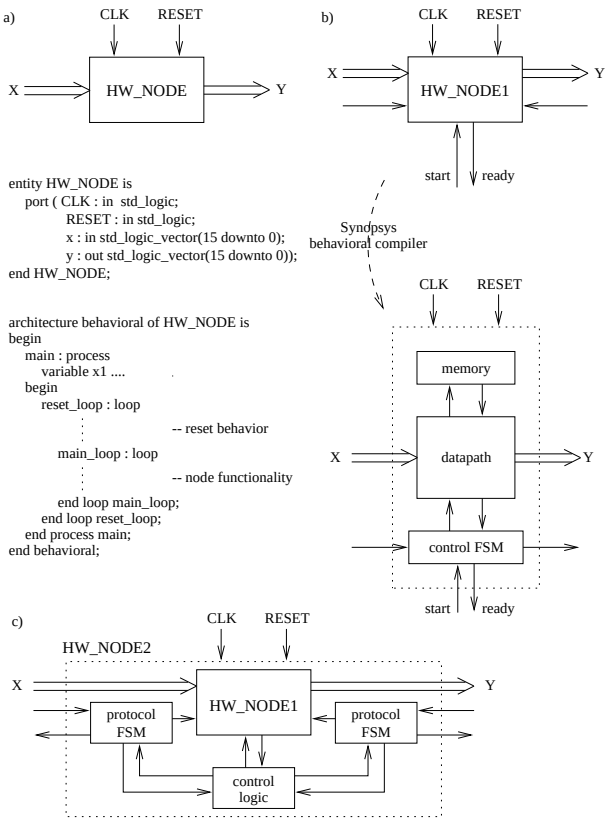

Figure 10: Implementation of a hardware node

an actor firing depends only on the satisfaction of its firing rule. An actor fires as soon as it is enabled. The resulting hardware schedule may be called *self-timed*.

For actors realized in software, actors mapped to the same physical resource (processor) must be executed sequentially. Different policies for scheduling ready actors can be distinguished in this case:

- *static fixed-order scheduling*: Here, the order in which actors are scheduled is given at compile-time (see, e.g., in Example 10).

- *static self-timed scheduling*: In this case, each processor cyclically checks the fireability of each actor assigned to it and, if fireable at that time, executes its code. Reads on input channels to check enable conditions are realized non-blocking in this case to simulate the self-timing behaviour. Contrary to the first case, the relative execution frequency of actors is not fixed at compile-time.

- *dynamic (run-time) scheduling*: Dynamic scheduling is more general than self-timed scheduling in that an operating system is generated as part of the software that manages actors as processes in queues. In this case, very general scheduling policies of how to schedule enabled actors can be implemented, e.g., fixed-priority scheduling algorithms, preemptive scheduling, etc.

2.6. Hardware/software synthesis tool

The code generation for nodes, channels and interfaces has been implemented in a synthesis tool written in C++ (see Fig. 11). By describing its class concept, its main advantage will become clear, namely an easy way to retarget the interface generation to other target architectures and protocols implementing dataflow models. It consists of two main parts: (1) the *synthesis classes*, and (2) a *command line interface*. The synthesis classes provide an object-oriented approach for code generation.

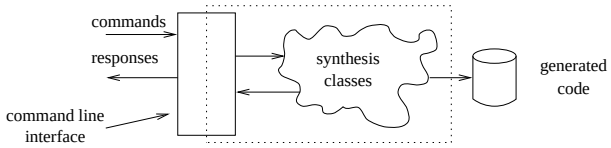

Figure 11: HW/SW synthesis tool

With the usage of abstract classes and inheritance, there exist code generators for the regular nodes and I_{SW} and I_{HW}. The tool is controlled by a command line interface. A command set has been specified for assembling dataflow graphs. With this interface, it is an easy task to couple the code generation tool to a higher level tool. It is planned to connect the tool to an optimization tool that outputs such a command file. Typical commands are explained in Example 6.

Figure 12 presents the class tree of the synthesis classes. Rounded boxes represent abstract classes, rectangles concrete ones. The triangles express an inheritance relation. The classes are split into three different levels:

- The *implementation independent* classes describe the basic properties of each graph component and have no code generation method.

- Given the partitioning in hardware and software, the code generators of the *language dependent* classes create VHDL for hardware and C for software components.

- Each *device dependent* class creates code for hardware interface I_{HW} and software interface I_{SW} for a dedicated microprocessor or microcontroller, e.g., the RISC microprocessor R3052.

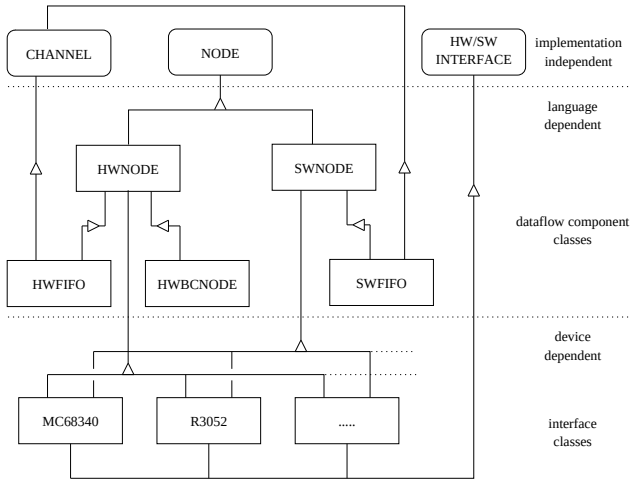

Figure 12: Class tree for synthesis

The abstract class `NODE` describes the implementation independent attributes, e.g., the number of node inputs. These properties are inherited by the classes `HWNODE` and `SWNODE`. They have additional methods which are necessary for the specific kind of implementation language. Examples are methods for protocol specification and for including system libraries. `SWNODE` is the code generation class that generates the function SW_NODE1 in Fig. 9b). The class `HWNODE` allows the generation of VHDL at the register-transfer level (RTL). `HWBCNODE` allows the behavioral specification of a hardware node which is on the same abstraction level as `SWNODE`. `HWBCNODE` implements all steps described in Section 2.4.4 to generate the entity HW_NODE2 in Fig. 10. `CHANNEL` specifies the basic properties of a channel like the depth of a FIFO. `HWFIFO` and `SWFIFO` have `CHANNEL` and `HWNODE`, or `SWNODE`, respectively as parent classes.

On the language dependent level, retargeting, e.g., moving hardware nodes to another FPGA type or moving software nodes to another microprocessor, is easily done. There are no modifications on the specifications necessary.

The next step towards the implementation is the generation of the hardware/software interfaces. The abstract class `HW/SW_INTERFACE`[1] describes the basic properties of different interface mechanisms, e.g., interrupts. The device dependent classes inherit the properties of the node classes. Each interface class has a built-in code generator for hardware and software, and generates the code for I_{HW}, and I_{SW}, respectively. Changing the implementation target means only switching to the appropriate device dependent class. The definition of a new device requires the creation of a new class and rewriting its code generation methods.

[1] Note that the hardware/software interface is not only necessary in case of a communication between a hardware and a software node, but may also be necessary in case of two communicating software nodes when both are bound to different physical processors.

2.7. A Test-platform for rapid prototyping

For our investigations in the field of hardware/software codesign, we built a heterogeneous test-platform (see Fig. 13 dotted rectangle) in the form of a PCI personal computer card consisting of a Motorola microcontroller MC68340 and two XILINX XC4025 FPGAs.

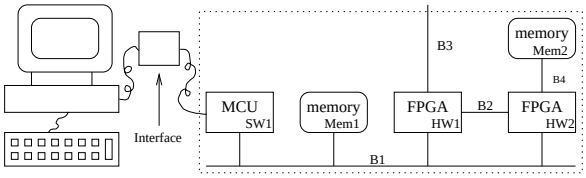

Figure 13: Test-platform for rapid prototyping

The two FPGAs HW1 and HW2 are directly coupled to the microcontroller unit SW1 via systembus B1. Between the FPGAs, a dedicated point-to-point bus B2 exists. Whereas the communication protocols of the microprocessor bus are dictated by the CPU and its bus specification, the bus B2 may be used to implement a user-defined bus and protocol.

Memory Mem2 together with bus B4 and HW2 may form a standalone microcomputer system with a dedicated user definable processor. Via PCI bus B3, the test-platform can communicate with other systems.

The platform is coupled to a workstation or a personal computer.

2.8. Case study

As a simple example, we present the mapping and implementation of the FIR-filter example in Section 1 to the target architecture in Fig. 1c). The target architecture consists of a software and a hardware computing resource SW1 and HW1, respectively. They are coupled via the microprocessor bus B1. There is the source node So producing data. For each input, the FIR-filter (node FIR) calculates an output which is then consumed by the sink node Si. The implementation process starting from the dataflow graph in Figure 1b) is divided into several steps.

2.8.1. Binding

First, the binding of nodes to computing resources and of edges to channels is done (manually at this time,

see Fig. 1c). Next, interface nodes are inserted into the original graph specification. These nodes model the hardware and software interfaces in the final implementation (see Fig. 14).

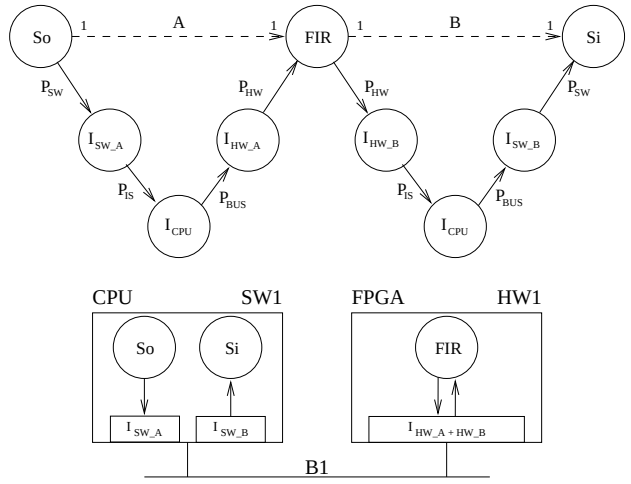

Figure 14: Mapped specification after introduction of interfaces

Example 6 *For the binding shown in Fig. 1, the hardware/software binding is expressed using our command-language interface:*

```
bind So =SW1;          bind Si=SW1;
bind FIR=HW1;          bind A =B1;
bind B  =B1;
```

The semantics of the commands should be self-explanatory.

2.8.2. Interface refinement

For each channel, different channel requirements may be specified.

Example 7 *A typical channel specification may be expressed by the following commands:*

```
set A.send=blocking; set B.send=blocking;
set A.recv=blocking; set B.recv=non_blocking;
set A.FIFO=0;        set B.FIFO=0;
set A.Size=integer;  set B.Size=integer;
```

For example, A is configured to be a non-buffered channel of integer datatype with blocking send and blocking receive semantics.

From each channel specification, a particular channel implementation will be generated.

Example 8 *We consider channel B in Fig. 14 for which memory mapped I/O with a data port d and a status port s is chosen (see Fig. 15). Via the data port, the hardware node can transfer data to the software node. The status port consists of one simple data ready bit R which is used for communication synchronisation. The flowcharts describe the semantics that the interfaces implement. The synchronisation 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 software node issues an explicit read access on the data port (dotted line). For SDF-graphs the termination condition of the loop is given in terms of the number of consumed tokens and the physical buswidth.*

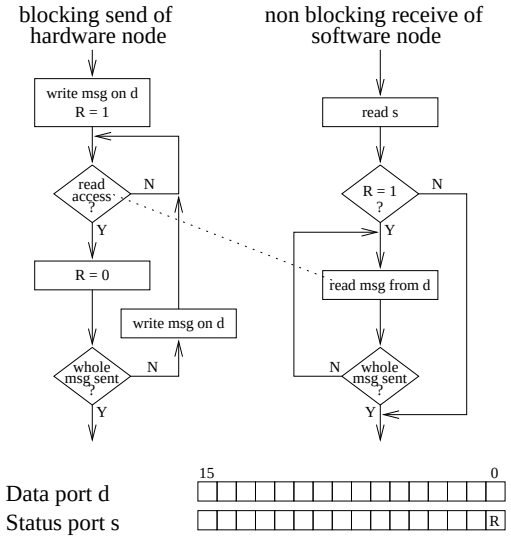

Figure 15: Interface refinement for channel B

2.8.3. Code generation

Example 9 *Figure 10 describes the basic steps to implement the FIR-node which is mapped to hardware. The top level of the FIR-filter (see Fig. 16) consists of the assembled entity FIR2 and the generated hardware interface. Finally, this assembly is compiled by logic synthesis. The result is a netlist at the level of logic gates.*

Example 10 *A code template for the software node So is shown in Fig. 17. a) The initial description of the node behaviour is given by a C-function. b) Now, a node function is generated for each software node that includes the communication primitives (sendA(),...) to read and send data. The semantics of the generated communication functions depends on the actual channel specification. c) The complete software program consists of the two functions So1(), Si1(), the generated*

interface functions (sendA(),recvB()), and a scheduler. Additionally, there is a main() procedure in which the initialization of the hardware (configuration of the FPGAs), the initialization of drivers, and the initialization of memories with initial data takes place. Then, main() calls the scheduler. d) The scheduler is created by the software synthesis tool, too. In the example, static-scheduling has been chosen (Fig. 17d).

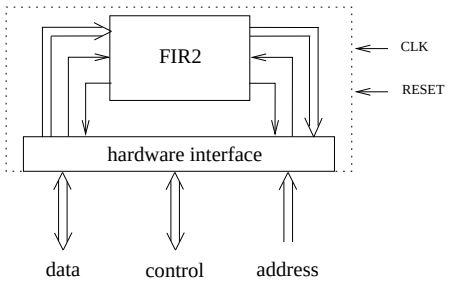

Figure 16: Implementation of the FIR hardware node

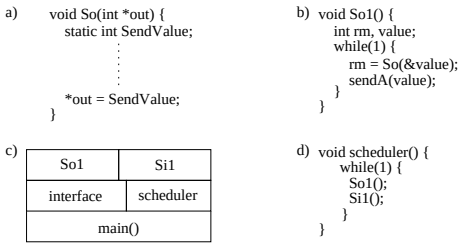

Figure 17: Implementation of the software partition of the FIR-filter

3. References

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

[2] G. Bilsen, P. Wauters, M. Engels, R. Lauwereins, and J. Peperstraete. Development of a static load balancing tool. In *Proc. of the fourth Workshop on Parallel and Distr. Processing*, pages 179–194, Sofia, Bulgaria, 1993.

[3] T. Blickle, J. Teich, and L. Thiele. System-level synthesis using Evolutionary Algorithms. *accepted J. Design Automation for Embedded Systems (Special Issue on Partitioning Methods for Embedded Systems)*, 1997.

[4] 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.

[5] 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.

[6] F. Commoner and A.W. Holt. Marked directed graphs. *Journal of Computer and System Sciences*, 5:511–523, 1971.

[7] J.B. Dennis. Data flow supercomputers. *IEEE Computer Magazine*, pages 48–56, 1980.

[8] 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.

[9] 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.

[10] R. W. Hartenstein, J. Becker, and R. Kress. Custom computing machines vs. hardware/software co-design: from a globalized point of view. In *Proc. 6th Int. Workshop on Field Programmable Logic and Applications, FPL'96. Lecture Notes in Computer Science, Springer Press*, Darmstadt, Germany, September 1996.

[11] IEEE. *IEEE Standard VHDL Language Reference Manual.* IEEE, IEEE STd. 1076-1987, 1987.

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

[13] 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.

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

[15] E.A. Lee and D.G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, 1987.

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

[17] I. Page and W. Luk. Compiling Occam into FPGAs. In W. Moore and W. Luk, editors, *FPGAs*, pages 271–283. Abingdon EE&CS Books, England, 1991.

[18] 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.

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

[20] G. C. Sih. Multiprocessor scheduling to account for interprocessor communication. Technical Report UCB/ERL 91/29, Ph.D dissertation, Dept. of EECS, UC Berkeley, Berkeley, CA 94720, U.S.A., April 1991.

[21] 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.

[22] 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.

[23] 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.