4

COSYNTHESIS AND COSIMULATION

In Chapter 3, algorithms to partition an application into hardware and software were described. Partitioning makes three attributes available for each node in the application: a hardware or software mapping, the type of implementation for this mapping, and a schedule.

The next step in the design process is to *synthesize* the mixed hardware-software system. The cosynthesis problem is to synthesize the hardware, software, and interface components for the final implementation. In Section 4.1 we discuss this problem. Based on the assumed target architecture, we develop an architectural model in Section 4.1.1. Our approach to synthesis is discussed in Section 4.1.2. We describe the particular mechanisms used to synthesize the hardware, the software, and the interface in Sections 4.1.3 to 4.1.5. Some of the other approaches to synthesis are discussed in Section 4.1.6.

Hardware-software *cosimulation* is the process of simulating the hardware and software components of a mixed hardware-software system within a *unified* environment. This includes simulation of the hardware modules, the processor, and the software that the processor executes. The cosimulation problem is discussed in Section 4.2. The requirements of a cosimulation environment are outlined. Ptolemy has an infrastructure that supports most of these requirements. The relevant details of the Ptolemy environment are discussed in Section 4.2.1. Our approach to hardware-software cosimulation within the Ptolemy environment is presented with the help of an example in Section 4.2.2.

4.1 Cosynthesis

After partitioning, each node of the DAG is annotated with a mapping, implementation bin, and schedule. We define cosynthesis as the problem of synthesizing the final implementation (hardware, software, and interface) from this annotated DAG.

4.1.1 Architectural Model

A particular target architecture for the mixed hardware-software system has been assumed in Chapter 2. As shown in Figure 4.1, the architecture consists of a single programmable processor and multiple hardware modules connected to a single system bus. Each node mapped to hardware is synthesized as a hardware module[1]. A hardware module consists of a datapath and controller, and input and

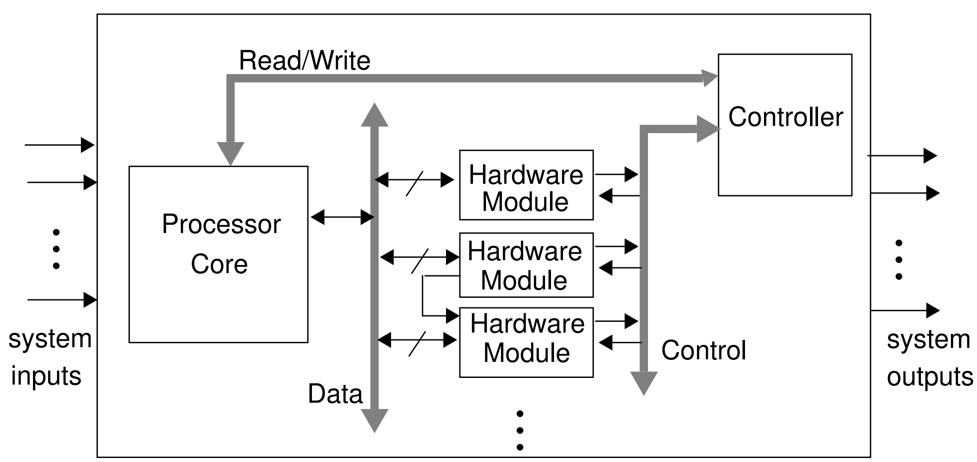


Figure 4.1. The target architecture.

output interfaces. The input and output interfaces are responsible for the communication between the hardware modules and the processor. The software component of the architecture is the program that runs on the programmable processor. This includes the communication code, i.e., the device drivers that manage communication of data between hardware and software. The architecture is assumed to be non-pipelined, i.e., one set of input data is processed completely before the second set arrives into the system. We assume that nodes in hardware and software communicate via a memory-mapped, asynchronous, blocking communication mechanism[2]. By *memory-mapped* communication between the software and hardware, we mean that data is communicated across the hardware-software boundary by writing to and reading from a shared address space. The various components in the architecture are configured as a *globally asynchronous locally synchronous* model. This means that each individual hardware module operates internally as a synchronous circuit, but the modules themselves communicate with each other and the processor asynchronously. In other words, within a hardware module, a module controller activates the various components of the datapath (ALUs, registers, multiplexers, etc.) at predetermined clock cycles[3]. Communication between a hardware module and the processor or between hardware modules is *asynchronous*, though. The reason is obvious. Since the schedule generated by partitioning is based on execution-time estimates, it is not guaranteed to be cycle-accurate. That is, we cannot exactly determine *when* a processor or hardware module should

---

1. We assume no hardware reuse *between* modules. This issue is further discussed in Section 4.1.3.
2. A system may contain several types of interfaces, such as serial ports or custom-designed peripherals. All nodes need not communicate via the same mechanism. An appropriate communication mechanism may be selected for the different data transfers, depending on various factors such as the size of the data (number of words and the bitwidth) and whether the data is communicated on-chip or off-chip. In this work we simplify matters by restricting all nodes to use the same type of communication mechanism.
3. The datapath and controller will be discussed in detail in Section 4.1.3.

clock. The generated schedule is valuable, however, in that it can reliably indicate the *order* in which nodes execute. A global controller activates the hardware modules in this order.

Architecture of a hardware module

The architecture of a hardware module is shown in Figure 4.2. We refer to the datapath and controller collectively as the *kernel* of a hardware module. Each kernel has two handshaking signals, *ready* and *completion,* in addition to the input and output data signals. Each input and output of the kernel is connected to a data latch. A latch has an input enable (IE) and an output enable (OE) control. The kernel begins its computation after it receives a *ready* signal, which indicates that valid data is available on all of its inputs. On finishing computation, the kernel raises a *completion* flag. The *ready* signal is a function of the input enable signals and the *completion* signal and is generated by combinational logic.

The details of the hardware-software, software-software, and hardware-hardware interface are described next.

Hardware-software interface

In a memory-mapped scheme, each input and output of a hardware module corresponds to a unique address in the shared address space of the processor. A traditional implementation of memory-mapped communication requires an explicit

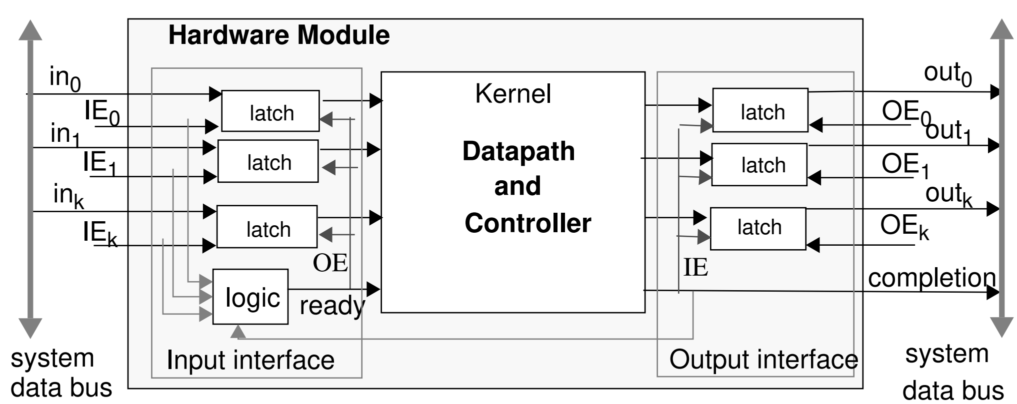


Figure 4.2. Architecture of a hardware module.

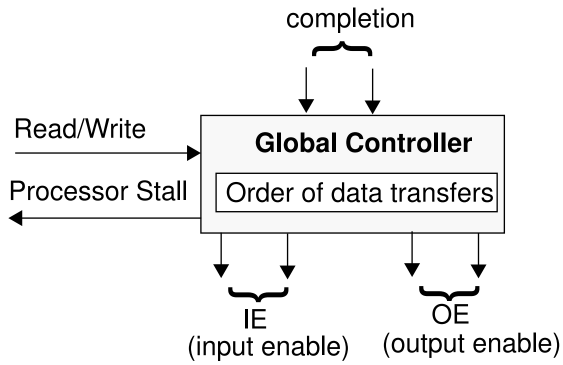


Figure 4.3. The global controller for the hardware-software interface.

address decoder that enables the appropriate input (output) of a hardware module when a write (read) to that address arrives. As the number of modules increases, the decoder tends to get quite large. To reduce the time overhead associated with semaphore checking and area overhead associated with explicit address decoders, we apply the *ordered transactions principle*, proposed by Lee *et al.* [Lee90] [Sriram93][4]. The hardware-software communication scheme works as follows. The schedule (determined by partitioning) is analyzed to determine the order of data transfers across the hardware-software interface. Each data transfer is assigned a unique memory address. Thus, the sequence of hardware module-addresses to which the processor sends read and write signals, is known apriori from the schedule. Each of these addresses corresponds uniquely to an input or output latch of a particular hardware module. A global controller, shown in Figure 4.3, uses the order information to activate the input and output latches of all the hardware modules. When the processor issues a *write* request, the global controller knows the corresponding input latch based on the order of transfers; it does not need to explicitly decode the address. Accordingly, it enables the corresponding input latch and the data from the processor is latched into the hardware mod-

4. They propose this principle in the context of shared-memory multiprocessor machines running SDF applications.

ule. Note that a hardware module's input data is always read before the next input arrives, since the architecture is non-pipelined. When all the inputs to a kernel are available, the combinational logic associated with the hardware module generates a *ready* signal that notifies the kernel to begin execution. When the processor issues a *read* request, the controller checks whether the corresponding hardware module has set its *completion* signal. If not, the controller stalls the processor until *completion* gets set. When *completion* is set, the controller enables the corresponding output latch and the data from the hardware module is made available to the processor. If the execution-time estimates are reasonably accurate, the ordered transactions approach does not introduce a significant overhead (the *ready* or *completion* signals are already set when the processor request arrives).

Software-software interface

Data transfers between two software nodes are assigned unique memory addresses in the internal data memory of the processor. Communication between two software nodes is achieved by writing the results to the corresponding locations in the internal data memory. Since the software executes sequentially according to the schedule, there is no need for semaphore checking.

Hardware-hardware interface

Communication between two hardware modules is achieved by directly connecting the output latch of the sending hardware module to the corresponding input latch of the receiving hardware module. After the sending hardware module finishes sending data, its *completion* signal enables the input latch of the receiving module. The *completion* signal of the sending module is also used in the combinational logic that generates the *ready* signal for the receiving module.

4.1.2 The Cosynthesis Approach

The cosynthesis problem involves synthesizing the following components:

1. the datapath and controller for each hardware module.

2. the program running on the programmable processor. This includes the code to read from and write to the shared memory locations for communication with the hardware modules.

3. the global controller, the input and output interfaces of the hardware modules, and the netlist connecting the controller to the various hardware modules and the processor.

Our approach to synthesis is shown in Figure 4.4. The application is specified as an SDF graph. As discussed in Chapter 2, the SDF graph is first translated to a directed acyclic graph representing data precedences. The DAG is an implementation-independent specification of the application, i.e., nodes are at a task level of granularity and have no commitment to a particular hardware or software implementation. The partitioning tool (discussed in Chapter 3) annotates each node of the DAG with a mapping, implementation bin, and schedule.

This annotated graph is then passed to a technology-dependent **retargeting tool**. This tool replaces each node in the DAG (and all the nodes within its hierarchy) with a technology-dependent representation corresponding to the mapping and implementation bin selected by the partitioning tool. By technology-dependent representation, we mean an assembly or C code representation for nodes mapped to software, and a VHDL or Silage representation for nodes mapped to hardware[5].

5. Note that the technology-dependent hardware description is still not the final implementation; the hardware implementation (at the architecture or layout-level) for a node is generated by highlevel synthesis tools that operate on this description. We assume such a two-tiered approach because it is unreasonable to expect full implementations of task-level elements to be available in a library, while library elements for Silage and VHDL descriptions are readily available within Ptolemy.

Furthermore, if different implementations are available for a node (for instance, cascade and transpose form IIR filters, or inlined and subroutine-based software representations), the retargeting tool selects the one corresponding to the implementation bin selected by the partitioning tool. In the case of hardware-mapped nodes, transformation and resource-level implementation options are also possible, as discussed in Chapter 1. Since these come into play only at a lower level, the corresponding technology-dependent description of a hardware-mapped node is anno-

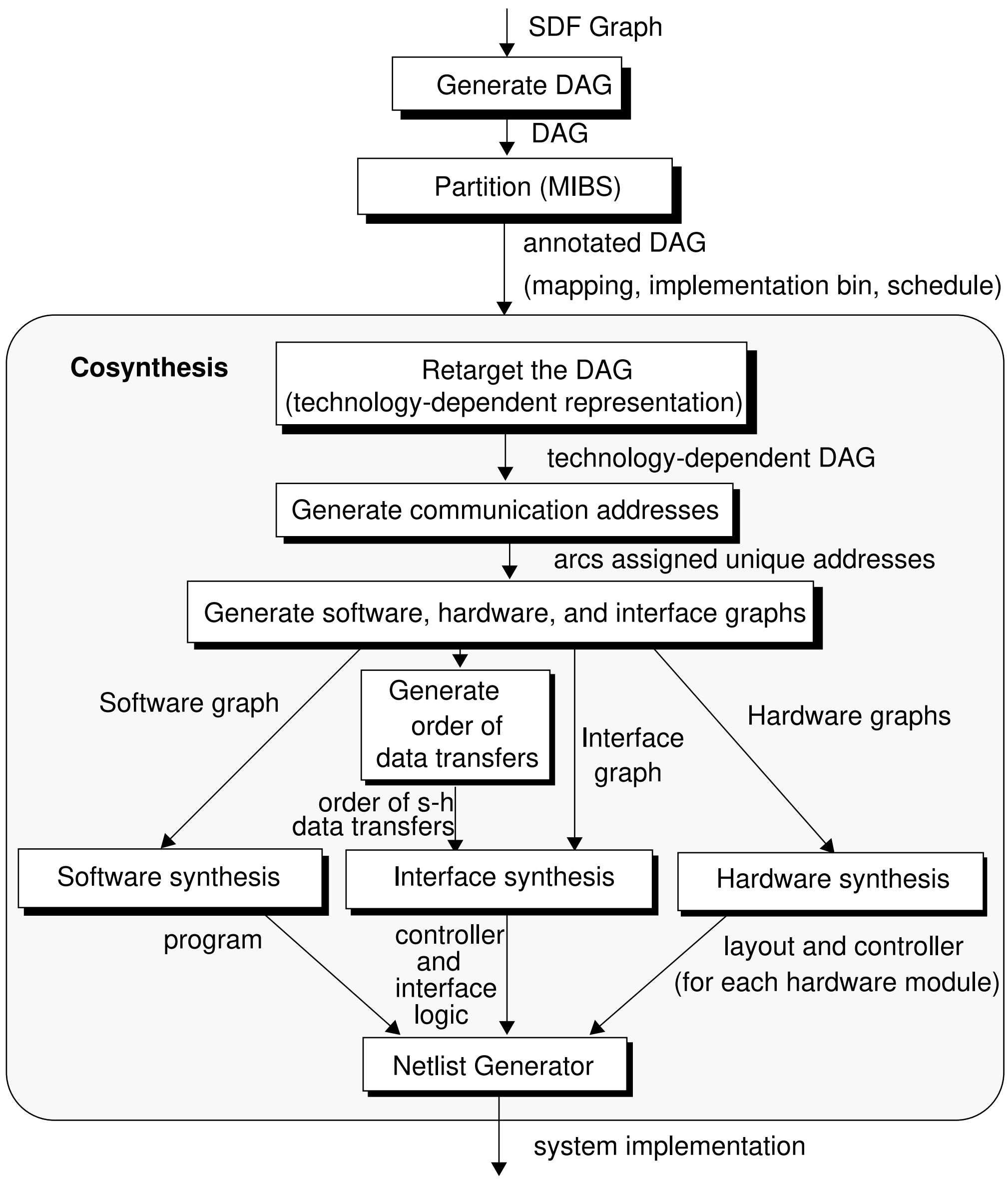


Figure 4.4. Cosynthesis approach

tated with the selected transformation and sample period. This information is then used by the hardware synthesis tools.

This retargeting approach assumes the existence of a library for each technology. For instance, suppose a node is an IIR filter. We assume descriptions of the IIR filter in C, VHDL, Motorola 56000 assembly code, etc. to be available. For instance, within the Ptolemy environment, extensive libraries for the various technologies, such as C, assembly code for various DSPs, VHDL, and Silage are available. Also, multiple representations (corresponding to different values of the implementation metrics) could be available for the same node within a given technology.

An alternative to the library-based retargeting approach is to *compile* the technology-independent representation of every node into its desired representation. In this approach, the node is described in a highlevel language, such as C. The C description for the node is then translated to an intermediate representation, which is typically a variant of a control-dataflow graph [McFarland90a]. The intermediate representation is *compiled* to either VHDL, or assembly code, depending on the desired synthesis technology. This compilation approach can be simplistically thought of as retargeting at the instruction level, accompanied by some traditional compiler-like optimizations. Such a compilation approach is used in the DSPStation [MentorGraphics1] for synthesizing hardware or software from a highlevel description based on Silage.

We have chosen the library-based retargeting approach. There are two advantages to this approach. First, it is computationally efficient and retains modularity. If a node (or its mapping or implementation) changes, the corresponding representation for the node is re-synthesized by simply selecting the new library element. In the compilation approach, the intermediate graph would have to be regenerated and compiled. Secondly, each module can be individually optimized (when added to the library), thereby improving the quality of the implementation. The disadvantage of this approach is that it depends on the richness of the library. If a corresponding element does not exist in the library, the user is forced to develop it.

After the retargeting step, each node in the DAG has been replaced by an equivalent technology-dependent representation. The **address generator** assigns a unique address to every data transfer involving a software and a hardware node, i.e., every arc connecting a software node to a hardware node is assigned an address in the processor's external address space.

The next step in the synthesis process is to generate the **hardware, software,** and **interface graphs**. Figure 4.5 shows typical hardware, software, and interface graphs for a simple example. These graphs are fed to the hardware, software, and interface synthesis tools respectively. A separate **hardware graph** is generated for each node mapped to hardware. Since a node could be represented hierarchically in the original description, the hardware graph can, in general, contain several subnodes. The hardware graph is annotated with the transformation and sample period corresponding to the implementation bin selected for this node. The hardware synthesis tool generates a datapath and controller for each hardware graph as required by its implementation bin. For software nodes, the synthesis technique is slightly different. All nodes mapped to software are combined in a single **software graph**. *Send* and *receive* nodes are added wherever a hardware-software communication occurs. Recall that the partitioning algorithm generates a global schedule that determines the order in which all the nodes in the DAG begin execution. The ordering between the nodes of the software graph is derived from this global schedule. The software synthesis tool generates a single program from the software graph. The program contains code for all the nodes in the software graph, where the code is concatenated in the predetermined ordering.

The **order generator** determines the order of transfers between nodes mapped to software and hardware. The interface synthesis tool generates the glo-

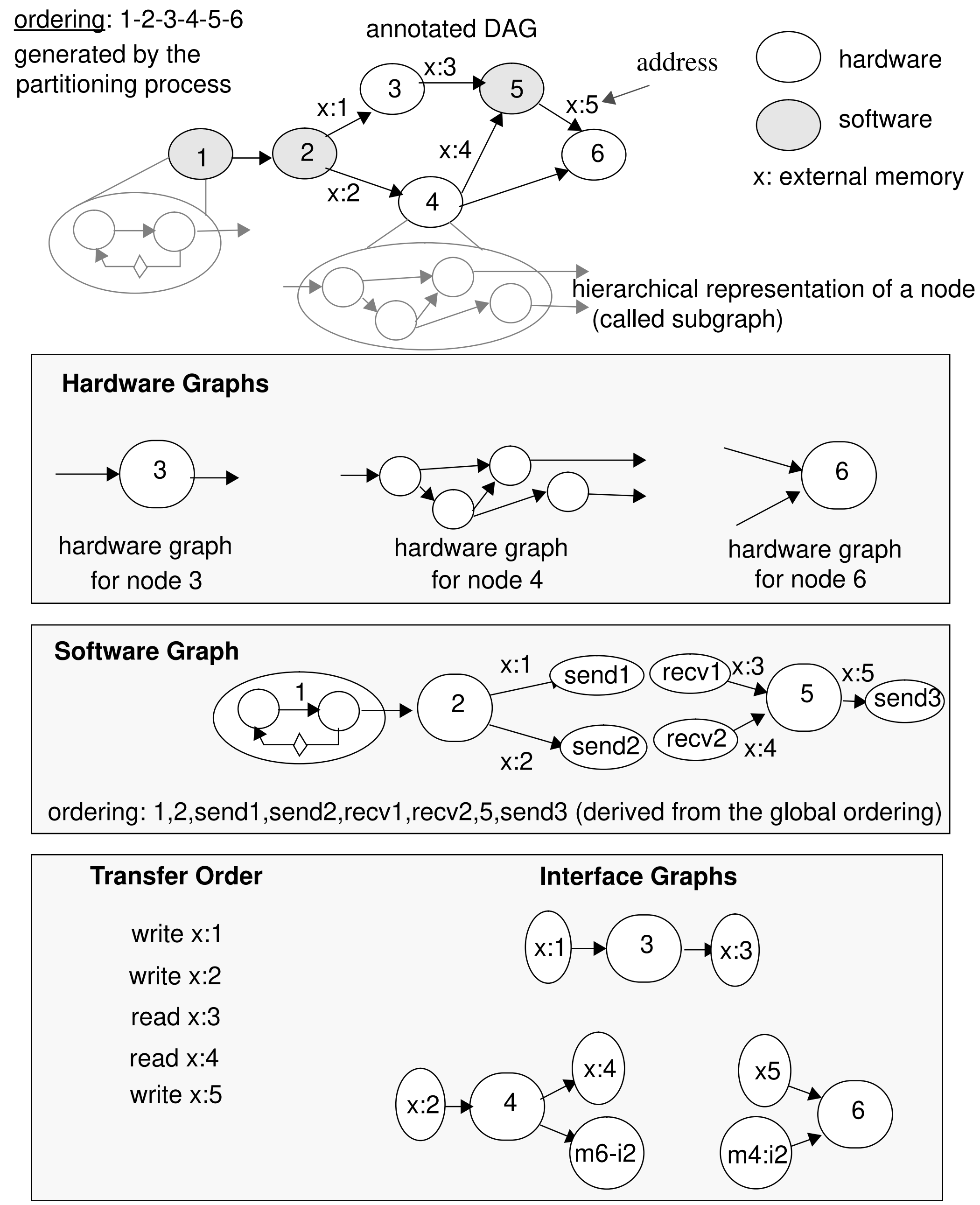


Figure 4.5. Hardware, software, and interface graphs. Each node in the hardware (software) graph contains a Silage or VHDL (C or assembly) description.

bal controller using this order of transfers. It also generates the interface glue logic (latches, combinational logic for generating the *ready* signal, etc.) for the hardware modules.

A netlist that describes the connectivity between the hardware modules, the processor, and the global controller is next generated. Standard placement and routing tools can then be used to synthesize the layout for the complete system.

In the following sections we discuss the hardware and software synthesis tools in more detail.

4.1.3 Hardware Synthesis

Given a hardware graph (for a node), we wish to synthesize its datapath and controller. The general approach for hardware synthesis is to:

1. Generate a synthesizeable description of the hardware graph. A synthesizeable description is typically in a language such as Silage or VHDL. Such a description is generated by combining the technology-dependent description of all the subnodes in the hardware graph.

2. Feed this description to highlevel hardware synthesis tools to obtain the implementation for the node. Several highlevel synthesis tools that operate on such synthesizeable descriptions already exist — our approach is to use them directly.

Figure 4.6 summarizes the hardware synthesis mechanism. We restrict ourselves to Silage[6] descriptions for the hardware graph, and assume that the Hyper

6. Silage [Hilfinger85] is an applicative language that is well-suited to expressing DSP applications. Its key features include: dataflow semantics, multirate operators, manifest iterations, signal precision specification, and single-assignment. Several research and commercial synthesis systems use Silage as the specification language (for instance, Hyper from UC Berkeley [Rabaey91], Cathedral from IMEC [Laneer91], and DSPStation from Mentor Graphics [MentorGraphics1]).

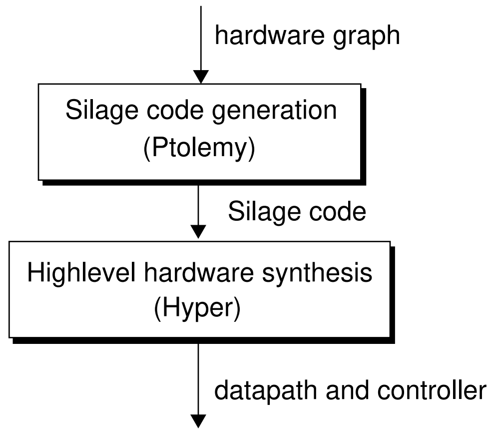


Figure 4.6. Mechanism for hardware synthesis for a single node

system [Rabaey91] is used to synthesize the datapath and controller from this Silage code.

Figure 4.7 illustrates the hardware synthesis mechanism with an example. Figure 4.7-a shows a simple task-level description (specified in Ptolemy), where one of the nodes is a 5th order IIR filter. Suppose that this node is mapped to hardware. Let us consider the synthesis of its implementation. Figure 4.7-b shows the hierarchical description of this node (a cascade of two biquads and a first order section). The first biquad is specified hierarchically in terms of elementary operations as shown in Figure 4.7-c, and the second biquad and the first order section are assumed to be monolithic. Figure 4.7-d shows the hardware graph for the node. Each subnode in the hardware graph contains Silage code. A single Silage program is generated by combining the Silage descriptions of the individual subnodes in the hardware graph. We have developed a Silage code generation mechanism, within Ptolemy, that generates Silage code starting from such a hardware graph (see [Kalavade93] and [Ptolemy95, Chapter 17] for details on Silage code generation[7]).

7. The Silage code generation process includes: (1) generating a schedule for the subnodes in the hardware graph (2) stitching together the code from the individual subnodes in the order specified by the schedule (this includes buffer management between the modules).

The generated Silage description for the IIR node in our example is shown in Figure 4.7-e. This description, along with the transformation and sample period values of the node (corresponding to its implementation bin) are used by Hyper to generate the implementation. Figure 4.7-f shows the final layout of the datapath and controller for the IIR filter.

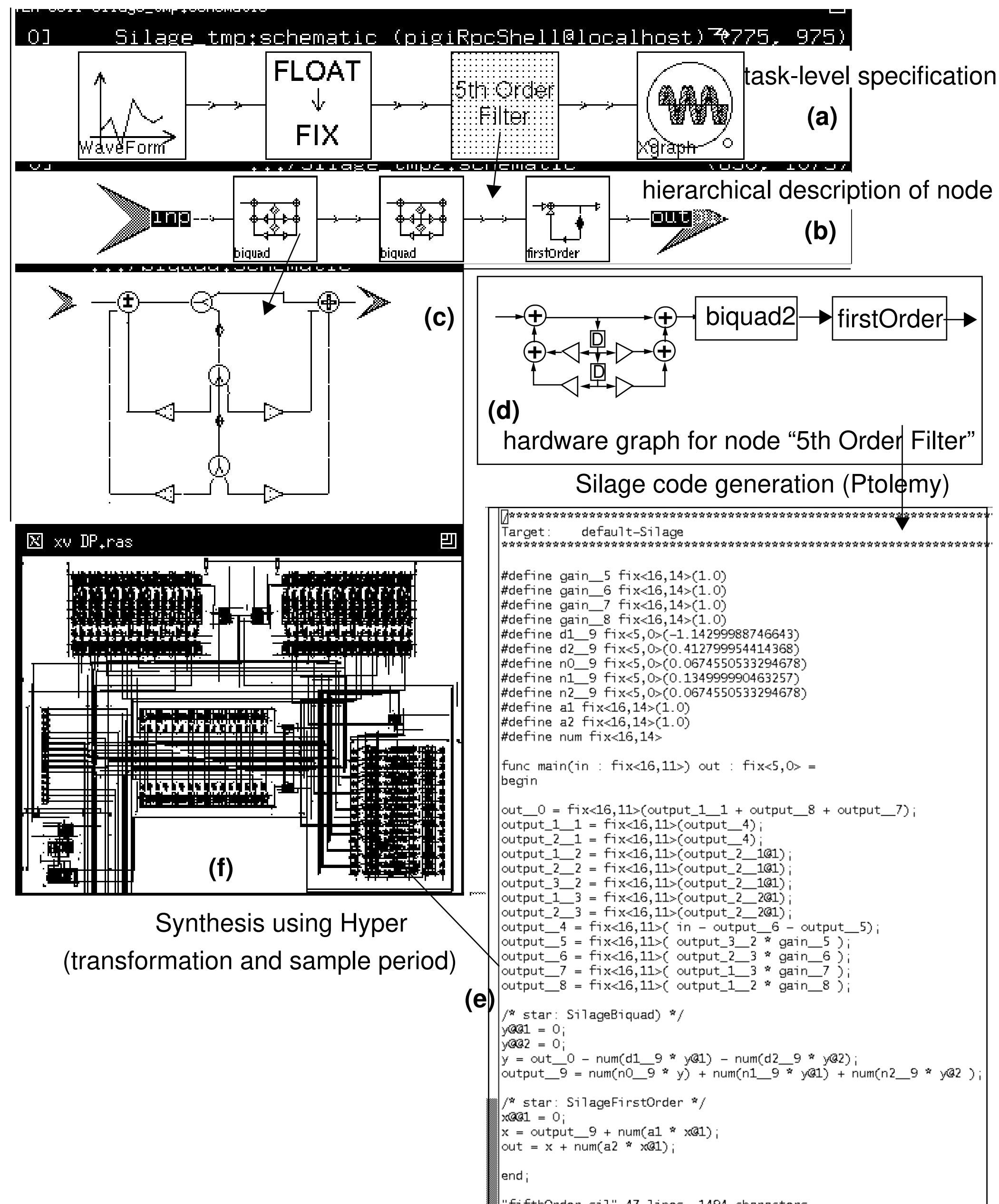


Figure 4.7. An example illustrating the hardware synthesis mechanism.

Hyper-generated Architecture

Figure 4.8 shows the generic architecture of the datapath and controller generated for a single hardware graph. The datapath comprises one or more execution units (ALUs, multipliers, adders, etc.). Inputs to the execution units are latched in registers. If an execution unit receives inputs from several sources, a multiplexor is added. Outputs of the execution unit are latched in tristate buffers. Data is communicated internally via a crossbar network. The datapath is controlled by a module controller. Associated with each execution unit is a local controller; the module controller controls the local controllers.

Note that the module controller generated by Hyper does not normally include the *ready* and *completion* signals. It can be modified slightly to incorporate

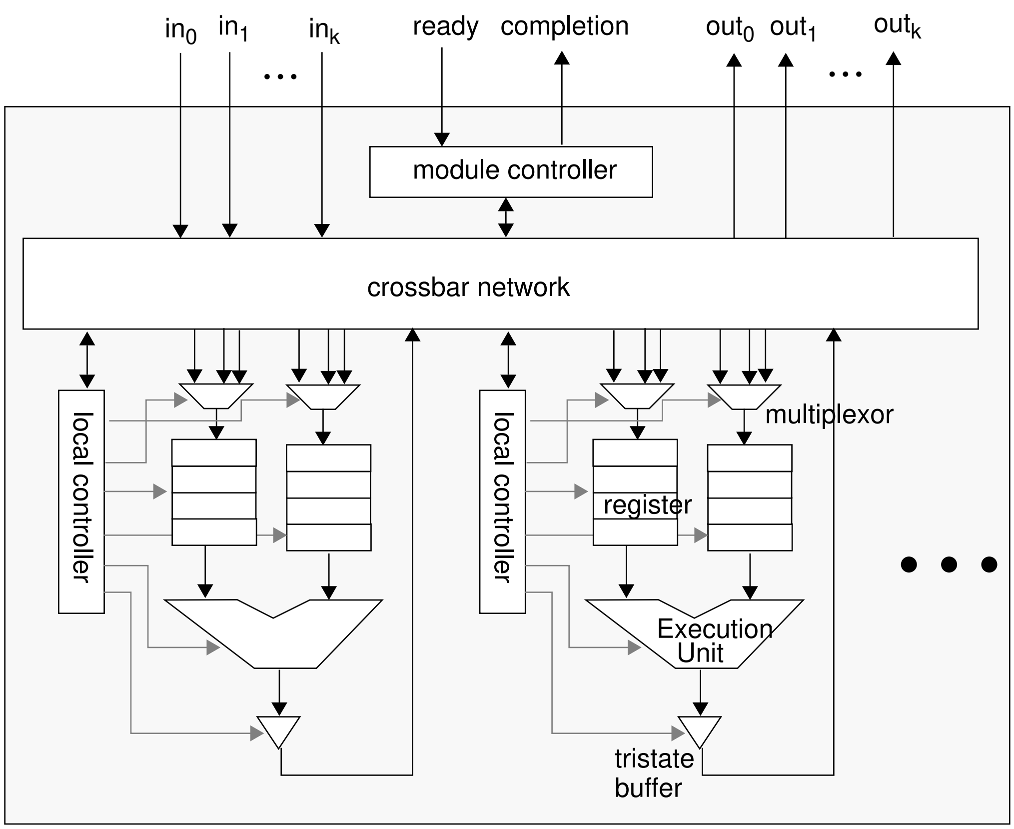


Figure 4.8. Architecture of the datapath and controller in a hardware module. This is the underlying architecture generated by Hyper, augmented with the *ready* and *completion* signals.

these signals. The module controller activates the execution units and the local controllers only after receiving the *ready* signal. It sets the *completion* signal after processing one set of inputs.

Hardware Reuse

A limitation of our current partitioning and synthesis mechanism is that it does not allow reuse of hardware *between* nodes. This is limiting especially in the case of multirate applications, where the DAG may contain several instances of a particular node. In such a case, the partitioning algorithm can be modified to take reuse into account[8]. The synthesis process would then have to take this reuse into account while generating the interface.

4.1.4 Software Synthesis

The software synthesis problem is to generate a software implementation (the program running on the programmable processor) corresponding to a software graph. Although the software synthesis technique described here applies to any target processor, our discussion assumes that the programmable processor used is a Motorola DSP 56000.

Recall that a software graph contains all the nodes mapped to software, augmented by *send* and *receive* nodes for communication. Each node in this graph contains a technology-dependent representation, i.e., a *codeblock* representing the functionality of the node. The software synthesis process essentially involves *stitching* together these codeblocks to generate a single program for the entire software graph. To preserve the functionality of the system, the codeblocks need to be put together according to the sequence generated by the partitioning tool. Note,

---

8. A simplistic approach is to consider the extremity and repeller measures of a node with multiple instances in the DAG and explicitly assign all the instances of the node to its preferred mapping.

however, that the schedule generated by the partitioning tool is at the level of nodes in the DAG; it does not contain any information on the sequence in which the subnodes *within* a hierarchical node should be executed. As a result, we first need to determine this sequence before we can stitch together the codeblocks. Such a sequence is generated by standard list scheduling techniques available within Ptolemy. The ordering of the nodes in the software graph is updated using this sequence. The final ordering is referred to as the *flattened ordering*. The individual pieces of code are then stitched together in the order specified by the flattened ordering. Note that this includes the code for the *send* and *receive* nodes too. The code for a *send* node contains a "write" instruction; it writes the results generated by the node connected to its input to the memory location assigned to its input arc. Similarly, a *receive* node contains a "read" instruction to read from the memory location assigned to its output arc. The end product is a program corresponding to the software graph. Our approach for software synthesis is summarized in Figure 4.9.

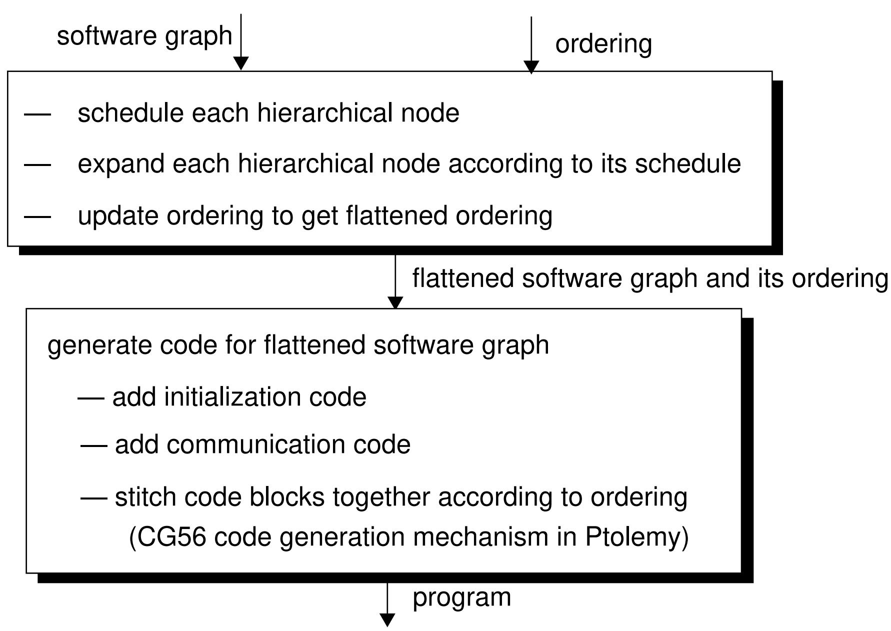


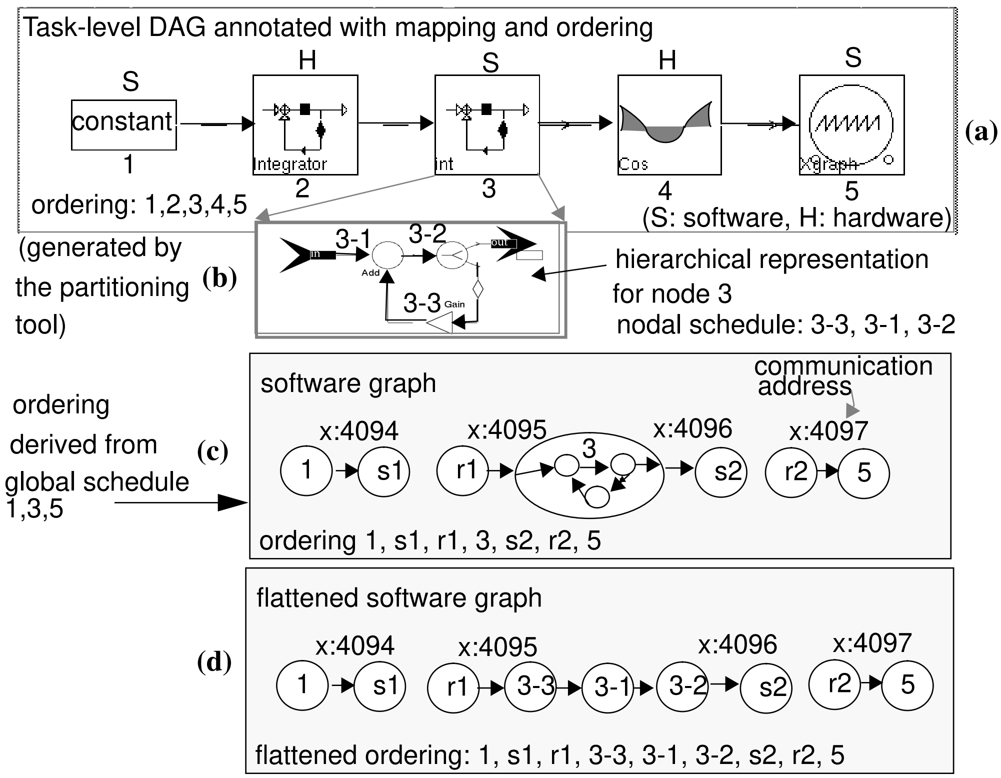
Figure 4.9. Mechanism for software synthesis from the software graph.

We have augmented the DSP 56000 code generation capability within Ptolemy ([Pino95a] and [Ptolemy95, Chapter 14]) to stitch together the code according to a *predefined* schedule. This predefined schedule is the order derived from the global schedule generated by the partitioning tool. The generated code is repeatedly executed by the DSP.

Figure 4.10 shows the software synthesis mechanism with the help of an example. Figure 4.10-a shows the task-level DAG annotated with the mapping and the ordering for a simple application (chirp signal generation). Suppose that nodes 1 (constant generator), 3 (integrator2), and 5 (xgraph) are mapped to software. Suppose further that node 3 is represented hierarchically as shown in Figure 4.10-b, while nodes 1 and 5 are monolithic. The software graph and the derived ordering is shown in Figure 4.10-c. The flattened software graph and its ordering are shown in Figure 4.10-d. The generated code (according to the flattened ordering) is listed in Figure 4.10-e.

4.1.5 Interface Synthesis

Interface synthesis involves synthesizing the global controller as well as the interface glue logic. A finite state machine description for the global controller can be generated using the order of transfers. An implementation for the controller can be synthesized using logic synthesis tools. The latches and other glue logic surrounding a hardware module can be synthesized using a template-based approach, such as that proposed by Sun *et al.* [Sun92]. The interface graphs can be used to generate a netlist connecting the various hardware modules. Currently, the interface synthesis is done manually, although the approach we have presented is relatively simple to automate.



(e)

```
; User:       kalavade
; Target:     scheduled-CG56
; Universe:   chirp
START
; initialization code from star Const1 (class CG56Const)
        org     x:1
        dc      0.00199997425079346

        org p:
;------------ main loop begins ------------------
LOOP
; code from star swcg1.Const1 (class CG56Const)

; code from star send1
        move    x:1,x0
        move    x0,x:4094

; code from star recv1
        move    x:4095,x0
        move    x0,x:2

; code for integrator2
; code from star Gain1 (class CG56Gain)
        move    x:4,x1
        move    #0.200000047683716,y1
        mpyr    x1,y1,a
        move    a,x:3
; code from star Add.input=21 (class CG56Add)
        move    x:2,x0  ; 1st input -> x0
        move    x:3,a           ; 2nd input -> a
        add     x0,a
        move    a,x:4           ; this move saturates
; code from star Fork.output=21 (class AnyAsmFork)

; code from star send2
        move    x:4,x0
        move    x0,x:4096

; code from star recv2
        move    x:4097,x0
        move    x0,x:5

; code from star swcg1.Xgraph1 (class CG56Xgraph)
        move    x:5,a
        move    a,y:1
        jmp LOOP
;------------ main loop ends ------------------
```

Figure 4.10. Software synthesis for chirp signal generator.

4.1.6 Other Synthesis Approaches

In this section we discuss some of the other approaches to cosynthesis. Systems that synthesize just hardware or just software from synthesizeable descriptions have been around for a while. A comprehensive summary of these tools exists elsewhere [Camposano91][Bier95a] and we will not discuss them here. Instead, we will discuss a few representative systems that focus on the higher-level cosynthesis problem.

Chiodo *et al.* [Chiodo94] focus on the cosynthesis of control-dominated applications. An application is typically described in Esterel or VHDL. This is translated to an internal representation consisting of a network of CFSMs (codesign finite state machines). CFSMs are similar to hierarchical concurrent FSMs. In addition, they communicate by broadcast. Events can have arbitrary propagation times, and do not necessarily follow the synchronous hypothesis. Each CFSM corresponds to a component of the system being modeled and is manually assigned to either hardware or software. Hardware is generated using logic synthesis tools. For software synthesis, each node in the CFSM is translated to a thread and a run-time scheduler controls the execution of threads. This is different from our approach, where a run-time scheduler is not required since the schedule is known at compile time. The key strength of the semantics of the CFSMs is that these systems can be verified using formal verification techniques.

In the VULCAN system [Gupta93], an application is specified in HardwareC and translated into a control-dataflow based internal representation. During software synthesis, the internal representation is compiled to C code. Hardware is synthesized by passing the internal representation through highlevel synthesis tools.

The COSYMA system [Henkel94] starts with a C-like description of the application, which gets translated to an internal syntax graph. After partitioning, the internal graph is compiled to HardwareC and C representations for hardware and software synthesis respectively.

Srivastava *et al.* [Srivastava91] have developed a framework, called SIERRA, for synthesizing the hardware and software components for multi-board systems. An application is described as a hierarchical network of communicating sequential processes. The user maps the application onto a system architecture, consisting of multiple boards with dedicated hardware modules and programmable processors. They have developed techniques that automatically generate the netlist for the system architecture. Software synthesis involves generating code for each process; a real-time kernel on the processors manages the software execution. The interface is synthesized using a template-based approach [Sun92].

Chou *et al.* [Chou92] address the problem of synthesizing the device drivers and glue logic for the hardware-software interface between microcontrollers and hardware devices. Given a list of device ports that need to be connected to the microcontroller and a list of available microcontroller ports, they present a procedure for assigning device ports to controller ports.

Coelho *et al.* [Coelho94] present an interesting variant of the cosynthesis problem. They address the problem of resynthesizing the system when the specifications change, while ensuring that hardware does not have to be resynthesized. They present techniques to resynthesize software such that the timing constraints imposed by the hardware are still met.

4.1.7 Summary

Cosynthesis is the problem of synthesizing the hardware, software, and interface components of the system, starting with a partitioned DAG. Our synthesis techniques target an architecture consisting of a single programmable processor and multiple hardware modules. The architecture is assumed to be non-pipelined and nodes mapped to hardware and software communicate using a memory-mapped, self-timed, blocking mechanism. Hardware-software communication is managed efficiently using the ordered transactions principle.

Our approach to cosynthesis is to decompose the partitioned DAG into hardware, software, and interface graphs, where each node in the hardware and software graphs is a technology-dependent representation of the original node. Pre-existing synthesis tools are used to generate the final implementation from these graphs. The emphasis in our work has been on generating synthesizeable representations for the hardware and software components. We use preexisting synthesis tools (such as Hyper and Ptolemy) to generate the final implementation.

4.2 Cosimulation

Hardware-software cosimulation is the process of simulating the hardware and software components of a mixed hardware-software system within a *unified* environment. This includes simulation of the hardware modules, the processor, and the software that the processor executes.

Hardware is typically simulated using discrete-event or cycle-driven simulators. The processor that executes the software can be modeled at various levels of detail, depending on the desired accuracy and simulation speed. In general, there is a spectrum of approaches for modeling a processor [Rowson94][Chang95][Becker92][Shanmugan94]. Possible approaches include:

1. Detailed processor models: In principle, the processor components could be modeled using a discrete-event model of their internal hardware archi-

tectures (datapath, instruction decoder, busses, memory management unit, etc.) as they execute the embedded software. The processor internals are modeled in a way typical of hardware systems, often using VHDL or Verilog. The interaction between models of individual processors and other components is captured using the native event-driven simulation capability supported by a hardware simulator. Unfortunately, most processor vendors are reluctant to make such models available, because they reveal a great deal about the internal processor design. Moreover, such models are extremely slow to simulate.

2. Bus models: These are discrete-event shells that simulate the activity on the periphery of a processor without executing the software associated with the processor. This is useful for verifying very low-level interactions, such as bus and memory interactions, but it is difficult to guarantee that the model of activity on the periphery is accurate; it is also difficult to simulate the interaction of the software with the hardware.

3. Instruction-set architecture models: The instruction set architecture can be simulated efficiently by a C program. The C program is an interpreter for the embedded software. It updates a representation of the processor state and generates events to model the activities on the periphery of the processor when appropriate. This type of modeling can be much more efficient than detailed processor modeling because the internals of the processor do not suffer the expense of discrete-event scheduling.

4. Compiled Simulation: Very fast processor models are achievable in principle by translating the executable embedded software specification into native code for the processor doing the simulating. For example, code for a

programmable DSP could be translated into Sparc assembly code for execution on a workstation. This is called *binary-to-binary translation*. The translated code has to include code segments that generate the events associated with the external interactions of the processor. In principle, such processor simulations can be extremely fast. The dominant cost of the simulation becomes the discrete-event or cycle-based simulation of the interaction between the components.

5. Hardware Models: If the processor exists in hardware form, the physical hardware can often be used to model the processor in a simulation. Alternatively, the processor could be modeled using an FPGA prototype, for instance, using Quickturn. The advantage of this sort of processor model is the simulation speed, while the disadvantage is that the physical processor must be available.

Thus, depending on the desired simulation accuracy and speed, one of these models can be used for simulating the processor in the mixed hardware-software system. A cosimulation environment should allow the processor to be modeled using any of these approaches.

It is often useful to be able to simulate the system at different levels of abstraction, throughout the entire design process. For instance, at the start of the design process, the hardware components may not have been synthesized completely. At this point, to study the interaction of the hardware and software, the hardware could be modeled functionally. As the design evolves, and more implementation-level details of the hardware become available, the functional model of the hardware can be replaced by a more detailed model at the netlist level. On the other hand, once the detailed operation of the hardware has been verified, one could go back and replace the hardware components by a highlevel model to increase the speed of the simulation. A cosimulation environment should support this migration across levels of abstraction.

Some of the components in the mixed hardware-software system may be off-the-shelf components, whose design is not part of the current design process. Such components can be modeled by a functional representation through the entire design process. It is not required to model the internal architectural details since they can be assumed to be correct, besides, we have no control over changing the model of such a component anyway.

In addition to the digital components, a system may also contain analog components. The cosimulation environment should also be able to support the modeling of such components and their interaction with the rest of the system. Figure 4.11 summarizes these aspects of cosimulation.

Thus it is clear that a single simulator, such as a discrete-event hardware simulator, cannot support all these requirements of hardware-software cosimulation. A framework that allows these different models to be mixed and matched is required for effective cosimulation.

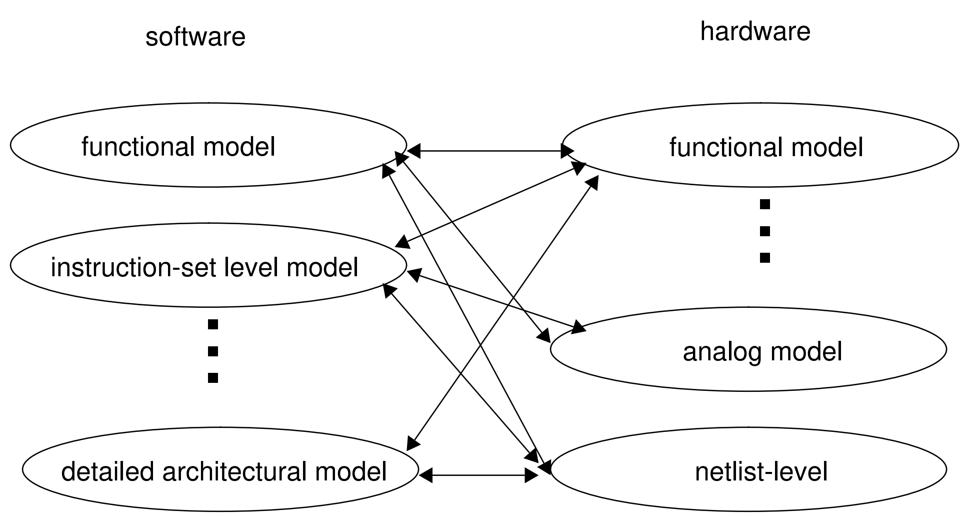


Figure 4.11. Cosimulation.

Ptolemy [Buck94a] is a heterogeneous simulation and prototyping environment, developed by the DSP Group at the University of California at Berkeley[9]. Ptolemy has the unique capability of supporting the integrated simulation of different models of computation. It meets most of the above-mentioned requirements of a cosimulation environment. We use Ptolemy for the cosimulation of mixed hardware-software systems.

In Section 4.2.1, we describe the features of Ptolemy that make it suitable for simulating systems with such diverse components. The use of Ptolemy for cosimulation is demonstrated with the help of an example in Section 4.2.2.

4.2.1 The Ptolemy Framework

Ptolemy is an environment for simulation and prototyping of heterogeneous systems. It uses object-oriented software technology to model each subsystem in its most natural and efficient manner, and has mechanisms to integrate heterogeneous subsystems into a whole.

Figure 4.12 shows the structural components of Ptolemy. The basic unit of modularity in Ptolemy is the Block. The system being simulated (or designed) is described as a network of Blocks. A Scheduler determines the operational semantics of a network of Blocks, i.e., it determines the order in which the Blocks are executed. The operation of the Scheduler is governed by the semantics of the underlying description. The lowest level (atomic) objects in Ptolemy are of the type Star, derived from Block. A Star is described by a C++ description. The description includes a function that gets invoked when a Block is initialized ("setup()"), a function to describe the run-time behavior of the Block ("go()"), and

9. While Ptolemy supports both simulation and synthesis of systems, we restrict the discussion in this section to the simulation aspects only. Complete details of Ptolemy can be found in [Ptolemy95]

a function that gets called at the termination of the simulation ("wrapup()"). A Galaxy, also derived from Block, contains other Blocks internally. A Galaxy may internally contain both Galaxies and Stars.

Ptolemy accomplishes the goal of multiparadigm simulation by supporting a plethora of different design styles encapsulated in objects called Domains. A Domain realizes a computational model appropriate for a particular type of subsystem. Some of the simulation Domains that are currently supported include 'Synchronous Data Flow' (SDF), 'Dynamic Dataflow' (DDF), 'Discrete Event' (DE), and the 'Digital Hardware Modeling Environment' (Thor).

A Domain in Ptolemy consists of a set of Blocks and Schedulers that conform to a common computational model — the operational semantics that govern how Blocks interact with one another. The Domain class by itself gives Ptolemy the ability to model *subsystems* differently, using a model appropriate for each subsystem. It also supports mixing these models at the system level to develop a heterogeneous system with different levels of abstraction. The mixture is hierarchical.

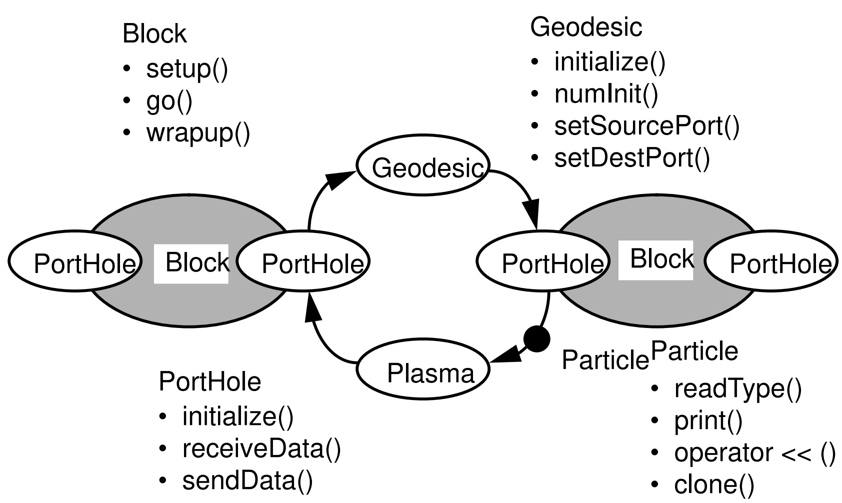


Figure 4.12. Block objects in Ptolemy send and receive data encapsulated in Particles to the outside world through Portholes. Buffering and transport is handled by the Geodesic and garbage collection by the Plasma.

Any model of computation can be used at the top level of the hierarchy. Within each level of the hierarchy, it is possible to have Blocks containing foreign Domains. This hierarchical heterogeneity is quite different from the concept of a *simulation backplane*, implemented for example in Viewlogic's SimBus system. A simulation backplane imposes a top-level model of computation through which all subsystems interact.

Figure 4.13 shows the top view of a system associated with a certain Domain called "XXX", associated with which are XXXStars and a XXXScheduler. A foreign subsystem that belongs to Domain YYY, and has its own set of YYYStars and a YYYScheduler is embedded in this XXXDomain system. This foreign subsystem is contained entirely within an object called an XXXWormhole. An XXXWormhole is a type of XXXStar — so at its external interface it obeys the operational semantics of the external Domain, but internally it consists of an entire foreign subsystem. A Wormhole can be introduced into the XXX Domain without any need for the XXXScheduler to know of the existence of the YYY Domain. The key to this interoperability is the interface between the internal structure of a

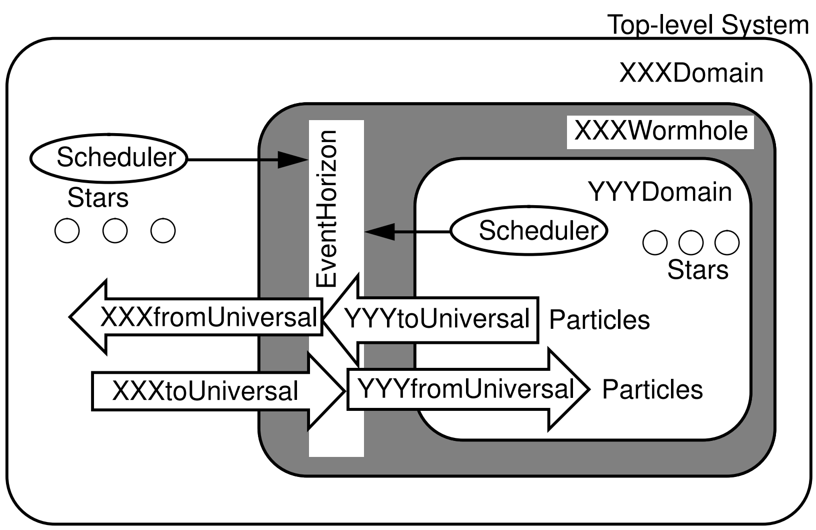


Figure 4.13. The universal EventHorizon provides an interface between the external and internal Domains.

Wormhole and its external environment. This interface is called the EventHorizon. The EventHorizon is a minimal interface that just supports exchange of data and permits rudimentary standardized interaction between schedulers. Each Domain provides an interface to the EventHorizon, and thus gains an interface to any other Domain.

The Domain and the mechanism of coexistence of Domains are the primary features that distinguish Ptolemy from otherwise comparable systems such as Comdisco's SPW and Bones, Mentor Graphic's DSPstation, and the University of New Mexico Khoros System [Rasure91].

We use the SDF and Thor simulation Domains for hardware-software cosimulation. Some details of these Domains are discussed next.

Synchronous Data Flow (SDF) Domain: The SDF Domain is a data-driven, statically scheduled Domain. "Data-driven" means that the availability of data on the inputs of a Star enables it. Stars with no inputs ("sources") are always enabled. "Statically scheduled" implies that the firing order of the Stars is determined only once during the start-up phase and this schedule is periodic. The SDF Domain supports simulation of algorithms, and also allows functional modeling of components such as filters and signal generators.

Thor Domain: The Thor Domain implements the Thor simulator [Thor86], which is a cycle-based register-transfer-level simulator for digital hardware. It supports the simulation of circuits from the gate level to the behavioral level. The Thor Domain makes it possible to simulate digital components ranging in complexity from simple logic gates to programmable DSP chips[10].

10. The Thor domain might likely be replaced in the future by a VHDL domain.

4.2.2 Cosimulation Using Ptolemy

Figure 4.14 shows an example of a mixed hardware-software system being simulated in Ptolemy. Figure 4.14-a shows the task-level SDF description of an application (telephone channel simulator) that is running on this system. Using the partitioning techniques described in Chapter 3, this application can be partitioned into hardware and software. The software synthesis mechanism described in Sec-

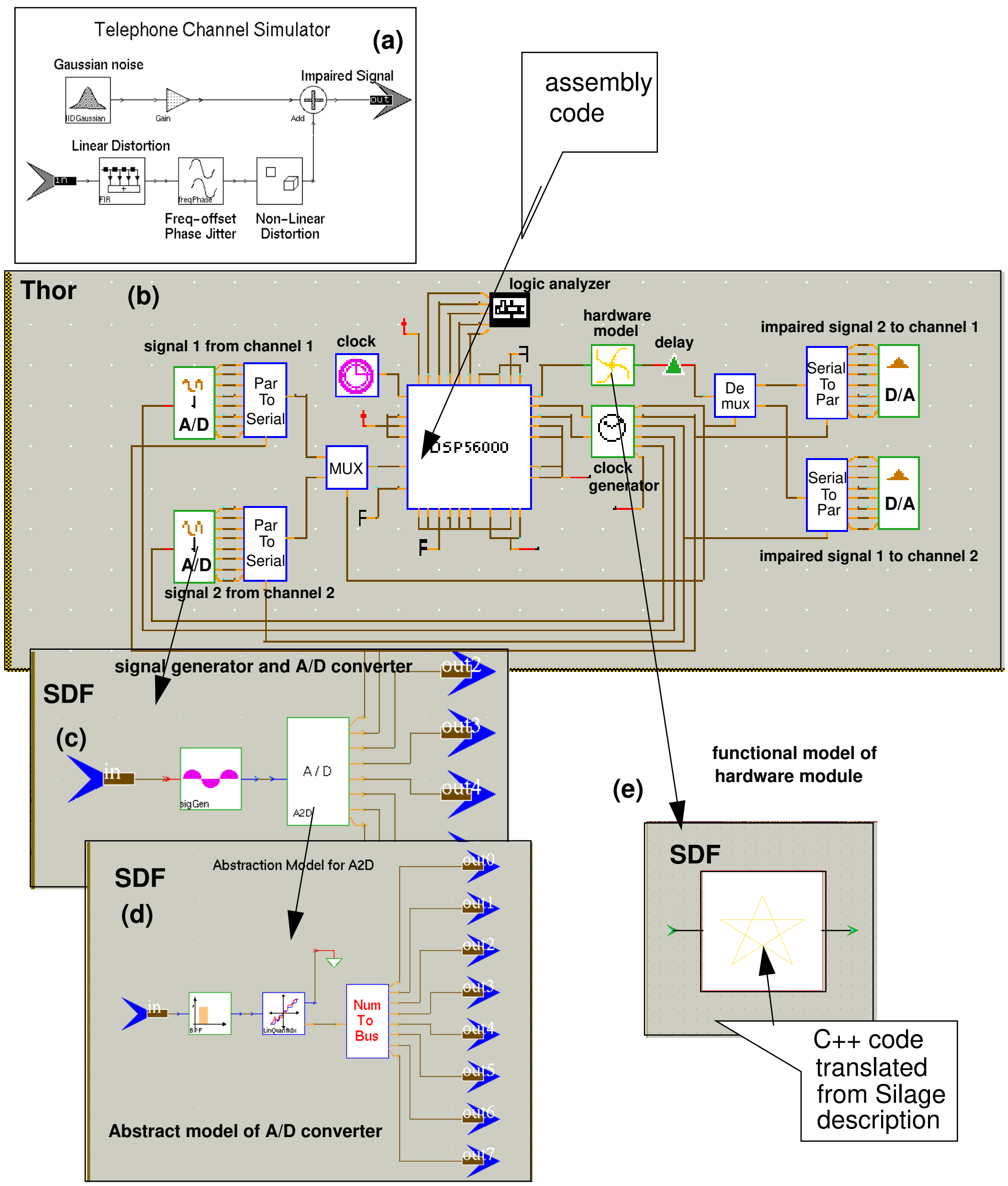


Figure 4.14. Hardware-software cosimulation using Ptolemy: system modeling

tion 4.1.4 is used to generate DSP assembly code corresponding to the nodes that are mapped to software. Hardware modules are synthesized for the nodes mapped to hardware using techniques described in Section 4.1.3. The next step in the design process is to simulate the interaction between the DSP (running the generated code) and the hardware modules.

4.2.2.1 System-level Modeling

The top-level model of the system is represented in the Thor Domain in Figure 4.14-b. The key components of the target architecture include the DSP, custom hardware, and the glue logic.

Modeling the DSP

The DSP56000 is modeled as a Star in the Thor Domain[11]. The "setup()" method of the DSP Star establishes a socket connection with *Sim56000*, Motorola's stand-alone simulator for the DSP56000. *Sim56000* is an instruction-set simulator; it accurately models the behavior of each of the processor's signal pins, while executing the code. During its "go()" method, this Star translates the logic values present at the processor's pins into values meaningful to the simulator, transfers them to *Sim56000*, and commands the simulator to advance the simulation by one step. It waits for the simulator to transmit the new logic values back to the processor pins and continues with the rest of the simulation.

Functional model of the hardware

Nodes mapped to hardware can be modeled at different levels of abstraction. In this example, we model the hardware at a *functional* level. The Silage code for the hardware module is translated to C++ code. This code is nested inside an

11. Early work on encapsulation of the Motorola simulator for interfacing with a stand-alone Thor simulation was done by Bier *et al.* [Bier89]. We have extended their techniques to encapsulate the simulator into a Ptolemy Star.

SDF Star and simulated as a Wormhole within the parent Thor Domain, as shown in Figure 4.14-e[12]. When this wormhole is run, it models the functional behavior of the hardware module. At the output of the hardware module is a “delay” block that emulates the execution time of this hardware module[13]. Such a model for the hardware module allows us to model the timing associated with the interaction between the hardware module and the rest of the system without having to go through a slow architectural simulation of the module.

Alternatively, the hardware could be simulated at a *structural* level within the same framework by replacing the functional model and the delay by a structural VHDL description of the hardware module.

Modeling the glue logic and the system I/O

The glue logic consisting of clock generators, decoders, and latches is easily modeled in the Thor Domain.

Besides processors and digital logic, it is often necessary to model analog components such as A/D and D/A converters, and filters that operate in conjunction with this digital hardware. These analog components can most conveniently be represented by their “functional models” using the SDF Domain. Often abstract functional modeling of components such as filters is sufficient — detailed behavioral modeling is not needed — particularly if an off-the-shelf component with well-understood behavior will be used in the final implementation. The Wormhole mechanism discussed earlier is used to mix the data-driven, statically-scheduled SDF models of analog components with event-driven, logic-valued Thor models of digital components within a single simulation. Figure 4.14-c shows a signal source being modeled as a sine generator followed by an analog to digital con-

12. Chapter 17 of [Ptolemy95] describes this Silage to C++ translation in some detail.
13. The execution time for a hardware module is determined by synthesizing an implementation for the node using hardware synthesis techniques discussed in Section 4.1.3.

verter. The A/D converter is modeled functionally as a bandpass filter followed by a quantizer, as shown in Figure 4.14-d. A functional model of this kind can be easily generated from the available device specifications.

4.2.2.2 System Simulation

Figure 4.15 shows the run-time behavior of the system. The logic analyzer

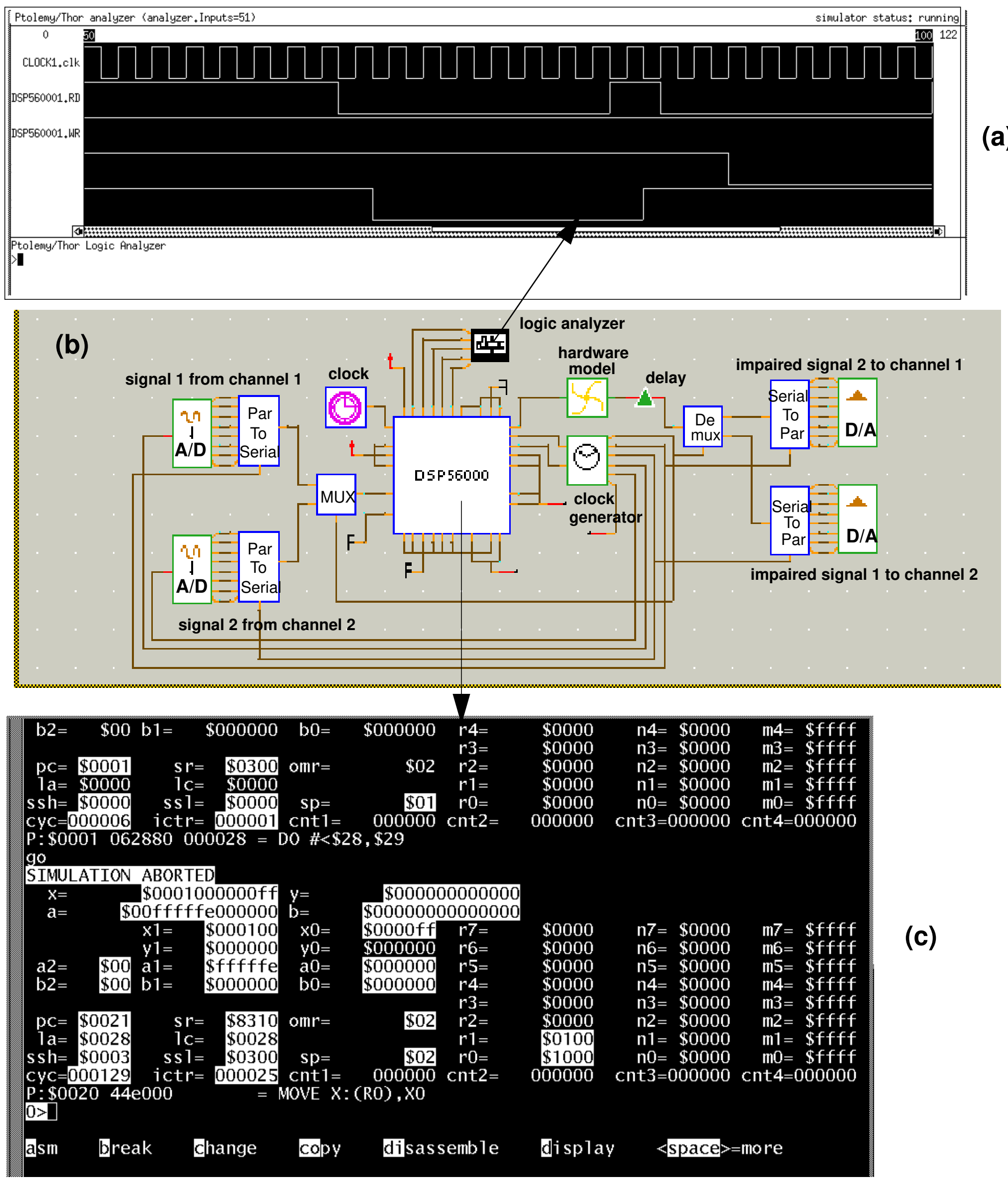


Figure 4.15. Hardware-software cosimulation using Ptolemy.

shown in Figure 4.15-a displays the traffic on the DSP bus. Figure 4.15-c shows the Motorola DSP simulator, which executes the generated assembly code. The simulation can be halted at any time by interrupting the simulator window and intermediate register contents can be examined. Data sent to the functional model of the hardware is transformed and passed onto the rest of the simulation after the delay equal to the execution time of that hardware module.

Thus the mixed hardware-software system can be simulated within the Ptolemy environment. The different domains are used to model the interaction between the components of the system at different levels of abstraction.