Presented at the NATO Advanced Study Institute Workshop
on Hardware/Software Codesign, Lake Como, Italy,
June 18 — 30, 1995.
(http://ptolemy.eecs.berkeley.edu/papers/effective)

EFFECTIVE HETEROGENOUS DESIGN AND CO-SIMULATION

W.-T. CHANG, A. KALAVADE, E. A. LEE
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720, USA

1. Introduction

In this chapter, we consider the problem of system-level design where subsystems are diverse. As a concrete example, consider an embedded system with the architecture shown in figure 1 (after [5]). The subsystems are implemented in both hardware and software, making this architecture a suitable candidate for hardware/software co-design. But even within the software portions, there is diversity. Control-oriented processes are mixed under the supervision of a multitasking real-time kernel running in the microcontroller. In addition, hard-real-time tasks run cooperatively on two programmable DSPs. The design styles used for these two software subsystems are likely to be quite different from one another, and testing

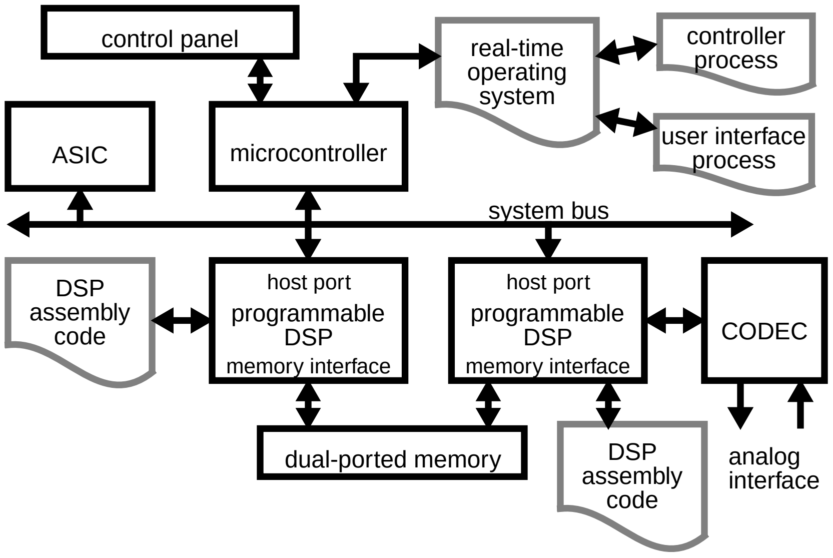


FIGURE 1. A typical embedded signal processing system.

the interaction between them is unlikely to be trivial.

Among the hardware subsystems in figure 1 is an ASIC, possibly designed using logic synthesis tools. On the other hand, a major part of the hardware design consists of interconnections of commodity components, such as processors and memories. Again, this time at the hardware level, we find diversity. The design styles used to specify and simulate the ASIC and the interconnected commodity components are likely to be quite different, and may not be supported by the same tools.

The system in figure 1, therefore, not only mixes hardware design with software design, but also mixes design styles within each of these categories. In an earlier paper [24], we observed that some research in system-level design seeks to unify these diverse styles.

> “Two opposing philosophies for system-level design are emerging. One is the unified approach, which seeks a consistent semantics for specification of the complete system. The other is a heterogeneous approach, which seeks to systematically combine disjoint semantics. Although the intellectual appeal of the unified approach is compelling, we have adopted the heterogeneous approach. We believe that diversity in design styles commonly used today precludes a unified solution in the foreseeable future. Combining hardware and software in a single system implementation is but one manifestation of this diversity.’’

In this chapter, we will examine the heterogeneous approach in more detail, concentrating on signal processing applications, and focusing on the idea that mixing models of computation is the key. We will consider system-level designs like those in figure 1, where commodity processors (possibly several kinds) are mixed with hardware designs (again possibly several kinds). We will not address the problem of joint design of an instruction-set architecture and the software that executes on it.

2. Component Subsystems

Commonly found components in embedded systems include:

Software or Firmware. It is rare to find a signal processing application with no software. At the very least, a low-cost microprocessor or microcontroller is used to manage the user interface. But it is common also to implement some of the core functionality in software, often using specialized programmable processors such as programmable DSPs.

Application-Specific Integrated Circuits. ASIC design has been the focus of design tools, even so-called “high-level’’ synthesis tools [13], for over a decade. The tools have developed to the point that certain systems can be synthesized fairly quickly. However, this approach to design is not always suitable. Complex, low-speed control functions are often better implemented in software. Moreover, many applications inherently require programmability, for example to customize the user interface. ASICs also cannot accommodate late design changes, and iterations of the design are expensive; thus they may not be suitable for implementation of immature applications. It is increasingly common to use one or more ASICs for the more well-understood and performance-intensive portions of the application, but to combine them with programmable processors to implement the rest, as shown in figure 1.

Domain-Specific Programmable Processors. Design re-use can drive down development time and system cost. For this reason, introducing enough programmability into a circuit to broaden its base of applications is often advisable. The suitable applications may range from a half dozen different algorithms to an entire domain such as signal processing. The processor itself can be designed by jointly optimizing the architecture, the instruction set, and the programs for the applications [34]. A major drawback to this approach is that it often requires an elaborate software infrastructure in order to make re-use viable.

Core-Based ASICs. This is an emerging design style, where programmable processor cores are combined with custom datapaths within a single IC. Manufacturers of programmable processors are making the cores of their processors available as megacells that can be used in such designs [6][30]. Alternatively, it is possible to use the core of an in-house processor [18]. Such core-based designs offer numerous advantages: performance improvement (due to critical components being implemented in custom datapaths, and faster internal communication between the hardware and software), field and mask programmability (due to the programmable core), and area and power reduction (due to integration of hardware and software within a single core). These designs are especially attractive for portable applications, such as those typically found in digital cellular telephony. The design of such systems requires partitioning the application into hardware and software, and exploring trade-offs in different implementations. A major drawback of this approach is that design tools do not currently support it well.

Application-Specific Multiprocessors. Some intensive applications have high enough complexity and speed requirements to justify development of an application-specific multiprocessor system. In such systems, the interconnect can be customized, along with the software and the selection of processors. Examples of design approaches for such systems range from homogeneous interconnections of off-the-shelf programmable components to heterogenous interconnections of arbitrary custom or commodity processors.

Other possible components include analog circuits and field programmable gate arrays. The issues in the design of such mixed systems range from partitioning of the algorithm between hardware and software, selection of the type and number of processors, selection of the interconnection network, synthesis of the software (partitioning, scheduling, and code generation), and synthesis of custom hardware.

The design styles involved in the above components are diverse. Tools that synthesize either complete software or complete hardware solutions are common within a single design style, but tools that support a mixture are rare.

3. Basic Definitions

3.1 MODELS OF COMPUTATION

Simulation of hardware designs is typically accomplished using a discrete-event simulator, such as that embodied in VHDL or Verilog simulators. In the discrete-event model of computation, a signal is a sequence of events where each event is tagged with a *time stamp*. The time stamp may be an integer, a floating-point number, or a data structure representing both the advance of time and the sequencing of *microsteps* within a time instant. In all cases, the job of the simulator is to sort events so that those with the earliest time stamps are processed first, and so that the events seen by any particular component have monotonically increasing time stamps. Time stamps, therefore, define a global ordering of events.

Discrete-event modeling is inherently expensive. The sorting of time stamps can be computationally costly. Moreover, ironically, although discrete-event is ideally suited to modeling distributed systems, it is very challenging to build a parallel discrete-event simulator. The global ordering of events requires much tighter coordination between parts of the simulation than would be ideal for parallel execution.

The relationship of the discrete-event model of computation to some related models is shown in figure 2. The discrete-event model is in part (d). The figure shows two signals, and illustrates that events in these two signals have a definite ordering relationship relative to one another. They are either simultaneous, or one precedes the other.

Another totally ordered model of computation is the discrete-time model, shown in figure 2(b). In this model, events occur synchronously, according to a *clock*. Again, events are unambiguously either simultaneous or one precedes the other. Unlike the discrete-event model, however, all signals have events at all clock ticks. This results in considerably simpler simulators, because sorting is not required. Simulators that exploit this simplification are called *cycle-based* simulators. Processing all events at a given clock tick constitutes a cycle. Within a cycle, the order in which events are processed may be determined by data precedences, which therefore define microsteps. Microsteps are also commonly used in dis-

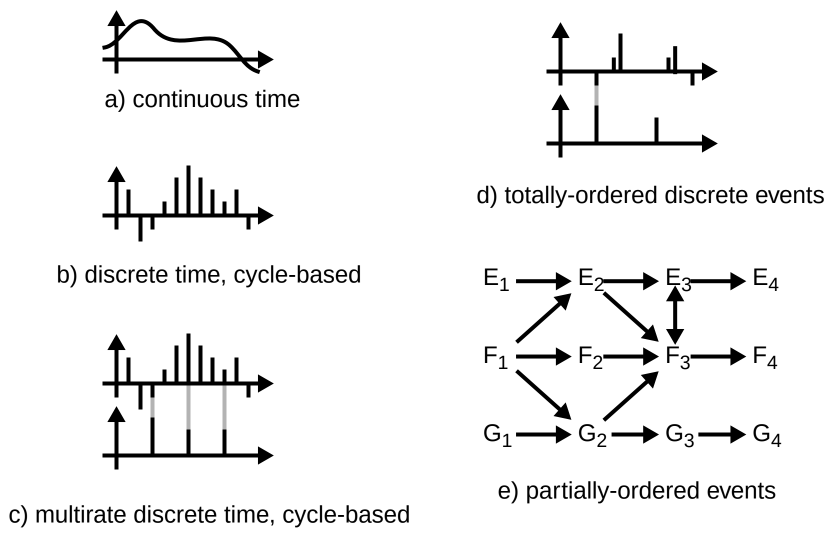

a) continuous time
b) discrete time, cycle-based
c) multirate discrete time, cycle-based
d) totally-ordered discrete events
E1 E2 E3 E4
F1 F2 F3 F4
G1 G2 G3 G4
e) partially-ordered events

FIGURE 2. A taxonomy of models of computation

crete-event simulators to control the order in which simultaneous events are processed.

The basic discrete-time model is inefficient for modeling systems where events do not occur at the same rate in all signals. While conceptually such systems can be modeled, using for example null tokens to indicate the absence of an event, the cost of processing such tokens is considerable. Fortunately, the discrete-time model is easily generalized as shown in figure 2(c) to multirate systems. Here, every *n*-th event in one signal aligns with the events in another.

The multirate discrete-time model is still somewhat limited. It is an excellent model for clocked synchronous circuits, and for synchronous signal processing systems, but in situations where events occur irregularly, it can again be inefficient.

A model that generalizes the multirate discrete-time model without paying the full price of a discrete-event model is embodied in the so-called *synchronous languages* [3]. Examples of such languages include Esterel [8], Signal [4], and Lustre [19]. In synchronous languages, a signal consists of a sequence of events that is conceptually (or explicitly) aligned with a *clock signal*. The clock signals define an ordering of events, so that any two events are either simultaneous (sharing the same clock tick) or one precedes the other. The compiler reasons about these ordering relationships and detects inconsistencies, contradictions, or incompleteness in the definitions.

Various looser models of computation specify only a partial ordering between events, as depicted in figure 2(e). This means that while events within any given signal are ordered, events in different signals may or may not have an ordering relationship. In figure 2(e), event F_1 precedes E_2, but E_2 and G_2 are *incomparable*. No ordering between them is defined. This type of specification has the advantage that it avoids *overspecifying* a design. If an ordering relationship is not important in a design, why specify it? Specifying it may severely constrain the implementation options. Thus, for example, while discrete-event simulators are difficult to parallelize, dataflow models, which are usually partially ordered [33], are comparatively easy to parallelize.

3.2 PROCESSOR MODELS

Simulation of systems like that depicted in figure 1 requires more than cycle-based or discrete-event modeling of circuits. The processors and the software they execute have to be modeled as well. The obvious approach to simulating such mixed hardware/software designs is detailed simulation or emulation of the hardware executing the software. The processor that executes the software can be modeled at the hardware architecture level or at the instruction set level [38]. Modeling it at the hardware architecture level can lend insight into the detailed operation of the processor on particular piece of software, but it is a very expensive approach. Modeling the processor at the instruction set level can be significantly faster, but still may be too slow for proper validation of a mixed hardware/software design. We explore these and other alternatives below.

Detailed processor models. In principle, the processor components could be modeled using a discrete-event model of their internal hardware architectures (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. This approach has the advantage that the processor model is often available as a natural consequence of the processor design cycle. 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. Even with a fairly abstract model of the processor in VHDL, it is optimistic to expect more than a few thousand instructions per second for a single processor simulator on a standard workstation. Besides, the level of detail is inappropriate if a commodity processor is being used. The processor is not under design, so its internal details need not be simulated. This is not the recommended solution.

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.

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. However, they still may not be fast enough for practical evaluation of a complete system design.

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 assembly code for execution on a workstation. This is called *binary-to-binary translation* and has been used by computer vendors for some time to port code from an older architecture to a newer one. In our context, the translated code has to include code segments that generate the events associated with the external interactions of the processor. Moreover, the operating system (if there is one) has to be simulated along with the application program, as done for a different reason in [11]. In principle, such processor simulations can be extremely fast, particularly if one sacrifices debug information. The dominant cost of the simulation becomes the discrete-event or cycle-based simulation of the interaction between the components. This still may be too slow for certain algorithmic evaluations.

Hardware Models. If the processor exists in hardware form, the physical hardware can often be used to model the processor in a simulation using either in-circuit emulators or FPGA prototypes such as Quickturn. The advantage of this sort of processor model is the simulation speed, while the disadvantage is that the physical processor must be available.

3.3 SYSTEM-LEVEL MODELS

As argued above, detailed simulation of the hardware (as it executes the software) is often too slow to be useful, except when evaluating low-level hardware interactions. An alternative is to mix multiple levels of abstraction. Every simulation should be constructed using the most abstract model that contains the details being tested.

For algorithmic evaluation, for example, functional models should be used. To assess real-time performance of embedded software (the number of cycles needed to execute the code), instruction-set modeling or compiled simulation should be used. The results of such simulations should be used to back-annotate the more abstract models, so that subsequent simulations can use these more abstract models. In the case of embedded software, for example, an instruction-set simulation could be used to extract a trace of timed interactions with the outside hardware. This trace could then be used in a bus model for more accurate simulation of the hardware.

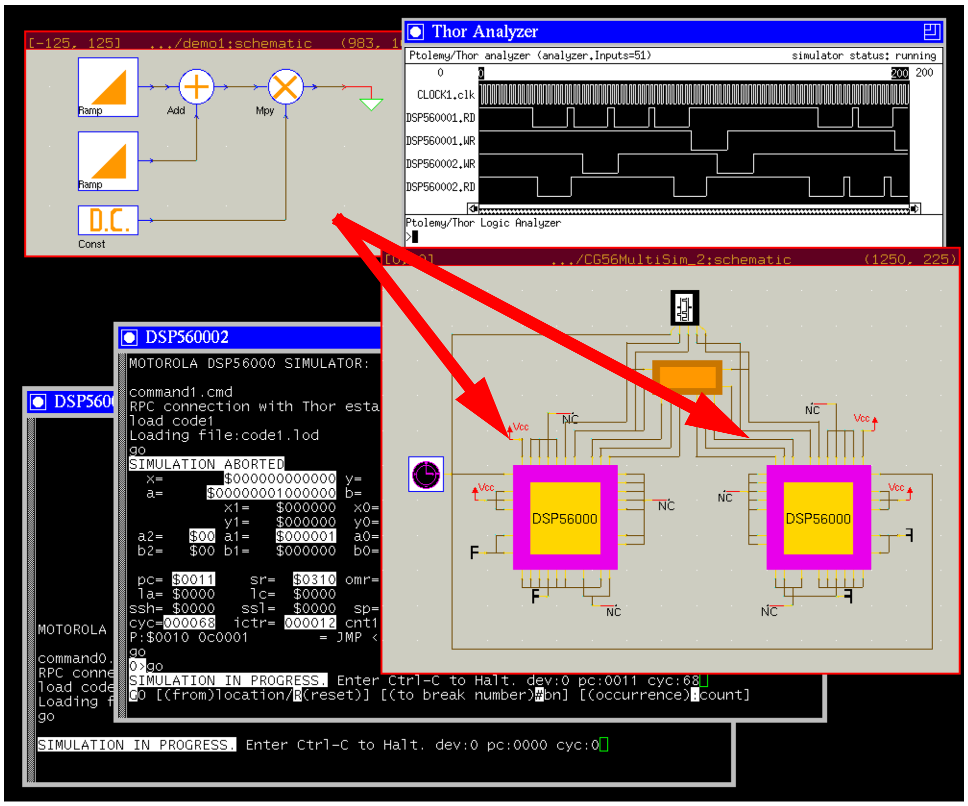
Abstract modeling of the software alone is too incomplete for accurate feedback on the cost and performance of a particular design. The situation is further complicated by the observation that high-level synthesis tools for both hardware and software components are themselves heterogenous. Some of the hardware might be designed at the register-transfer level, while other parts are designed using synthesis from more abstract representations.

Use of more abstract and specialized tools and languages for software design, particularly for embedded systems, is also increasing. Visual dataflow programming environments, for example, are common in the signal processing community. Hierarchical finite-state machine languages are catching on for control. And symbolic processing languages are used widely in scientific computing.

3.4 BASELINE CO-SIMULATION

As a concrete example of the capabilities minimally required to simulate systems like that in figure 1, consider the co-simulation shown in figure 3. This figure shows a board-level hardware design with two programmable DSPs. Each DSP is simulated using an instruction-set architecture model provided by the vendor of the DSP. The programmer's view into the state of the DSPs is shown in the windows at the lower left. The interaction between DSPs is simulated using a discrete-event register-transfer-level hardware simulator. A logic analyzer at the top shows the interactions of the DSPs with a dual-ported shared memory. At the upper left is a block diagram that specifies the software that is jointly executed on the two DSPs. That block diagram has dataflow semantics and is automatically partitioned for parallel execution.

The model in figure 3 is certainly heterogenous, and it represents capabilities that are starting to appear in commercial software environments for signal processing. The particular implementation shown in figure 3 is in the Ptolemy software environment from Berkeley [10]. However, this is still not nearly complete. It does not embrace the multiplicity of design styles that will be used in systems like that in figure 1, for example hierarchical finite-state machines for the embedded control code that runs in the microcontroller. It also does not embrace the


[-125, 125] .../demo1:schematic (983, 1
Ramp
Add
Mpy
Ramp
D.C.
Const
Thor Analyzer
Ptolemy/Thor analyzer (analyzer.Inputs=51)
simulator status: running
0
200
CLOCK1.clk
DSP560001.RD
DSP560001.WR
DSP560002.WR
DSP560002.RD
Ptolemy/Thor Logic Analyzer
.../CG56MultiSim_2:schematic (1250, 225)
Vcc
NC
DSP56000
DSP56000
DSP560002
DSP560

```
MOTOROLA DSP56000 SIMULATOR:

command1.cmd
RPC connection with Thor esta
load code1
Loading file:code1.lod
go
SIMULATION ABORTED
  x=        $000000000000 y=
  a=      $00000001000000 b=
              x1=   $000000  x0=
              y1=   $000000  y0=
 a2=    $00 a1=   $000001  a0=
 b2=    $00 b1=   $000000  b0=

 pc= $0011     sr=  $0310 omr=
 la= $0000     lc=  $0000
ssh= $0000    ssl=  $0000  sp=
cyc=000068  ictr= 000012 cnt1
P:$0010 0c0001          = JMP <
go
0>go
SIMULATION IN PROGRESS. Enter Ctrl-C to Halt. dev:0 pc:0011 cyc:68
GO [(from)location/R(reset)] [(to break number)#bn] [(occurrence):count]
```

```
MOTOROLA

command0.
RPC conne
load code
Loading f
go

SIMULATION IN PROGRESS. Enter Ctrl-C to Halt. dev:0 pc:0000 cyc:0
```

FIGURE 3. Baseline co-simulation system implemented in Ptolemy.

multiplicity of design styles that are used for hardware, such as high-level synthesis and logic synthesis. Such broader capabilities continue to mature within the Ptolemy environment [26].

4. Heterogeneous Simulation

In figure 4, the low-level co-simulation of the type implemented in figure 3 is shown in the bottom box. Models of the component processors interact with models of the rest of the hardware as they execute the embedded software. It is widely recognized that this type of co-simulation is needed in practical design environments. Not nearly so widely recognized is the need for a broader type of co-simulation that invites more abstract subsystem representations.

At the top of figure 4 are four classes of semantic models that can be used for abstract, system-level models of computation. Discrete-event, which is used for low-level hardware modeling (at the register-transfer level), can also be used for high-level system modeling. Discrete-event models are commonly used, for

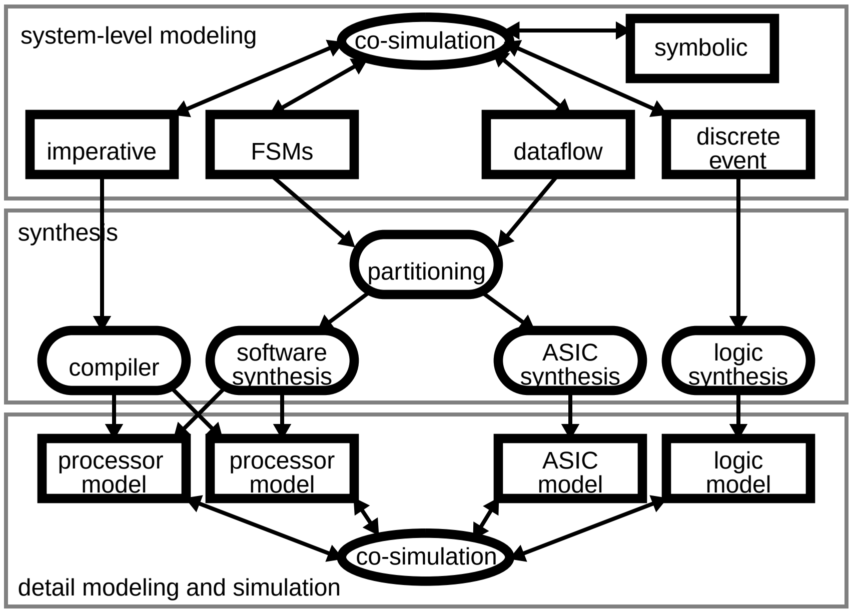


FIGURE 4. Models of computation and co-simulation.

example, to simulate communication networks, even on a global scale. Discrete-event models, however, are not well suited to specifying software designs because of their inherently slow execution. Hence, in figure 4, the discrete-event models are shown passing through only a hardware synthesis path, not a software synthesis path.

For signal processing, block diagram systems using dataflow semantics have grown in popularity. Dataflow (in most implementations) is a partially ordered model of computation [33]. It can be used to synthesize either embedded software [35][39], hardware [16][36][42], or both [24][25], with automatic or manual partitioning.

A similarly versatile class of computational models is the recently popular hierarchical finite-state machines, such as the statecharts model [20] and at least 20 variants [41]. Like dataflow, these have also been shown to be amenable to both hardware and software synthesis [21].

A fourth class of models of computation is the imperative model, such as that found in familiar languages such FORTRAN, C, Lisp, and the object-oriented languages such as C++ and Smalltalk. In this model of computation, a specification gives a total ordering on actions, not just events, and thus may be grossly overspecifying a concurrent implementation such as a circuit. Thus, although there have been a few experiments in hardware synthesis from imperative models, imperative models are mostly used to specify software.

The key observation is that a given system design is likely to involve subsystems where the best models of computations differ. Thus, some mixture of the four classes at the top of figure 4 might be used in the same design. Thus, co-simulation is required even at this high level of abstraction.

For signal processing, a background of symbolic manipulation precedes most detailed designs. This symbolic manipulation, which might use tools such as Mathematica or Maple, should be considered part of the design process and should be captured in the design flow [17]. It constitutes a fifth class that must be co-simulated, shown at the top right of figure 4.

In this chapter we take a broad view of the problem of heterogeneous design and simulation. We take as an assumption that compilation of the complete design down to a single, unified, detailed representation is not the solution. To cover all possible cases, this representation has to be far too detailed to allow for effective simulation. Instead, the problem boils down to one of finding clean ways to mix diverse models of computation, at varying levels of abstraction.

The problems posed are not simple, and have only been solved for certain special cases. We will use as examples of high-level representations: dataflow models (typically used for signal processing and numeric computation), hierarchical finite-state machines (typically used for control), and discrete-event systems (typically used for modeling the timing of hardware systems at varying levels of abstraction). We explain each of these models of computation, and give examples of design environments that support them. We then give examples of mixtures of such models, and use these examples to illustrate some of the fundamental problems that arise (together with a few solutions).

5. Multi-paradigm Design

We consider in this section mixtures of the classes of semantic models in figure 4. We begin with some background on each.

5.1 DATAFLOW PROCESS NETWORKS

In dataflow, a program is specified by a directed graph where the nodes represent computations and the arcs represent streams of data. The graphs are typically hierarchical, in that a node in a graph may represent another directed graph, and are often represented visually. The nodes in the graph can be either language primitives or subprograms specified in another language, such as C or FORTRAN. In the latter case, we are already mixing two of the models of computation from figure 4. Dataflow serves as a *coordination language* for subprograms written in a *host language*.

Some examples of graphical dataflow programming environments intended for signal processing (including image processing) are Khoros, from the Univer-

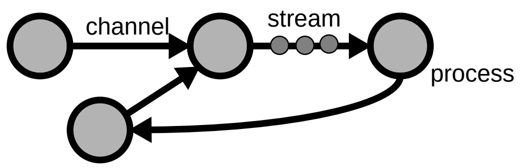


FIGURE 5. A process network.

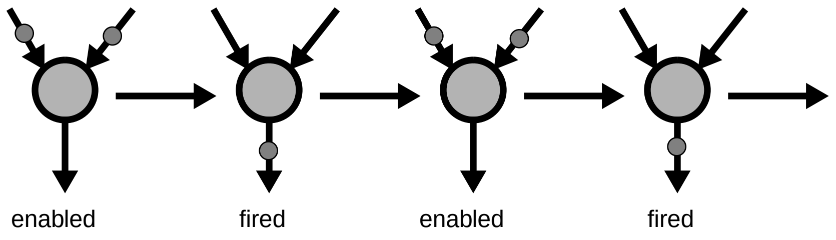


FIGURE 6. A dataflow process.

sity of New Mexico [37] (now distributed by Khoral Research, Inc.), Ptolemy, from the University of California at Berkeley [10], the Signal Processing Worksystem (SPW), from the Alta Group at Cadence (formerly Comdisco Systems), COSSAP, from Synopsys (formerly from Cadis), and the DSP Station from Mentor Graphics (formerly from EDC).

These software environments all claim variants of dataflow semantics, but a word of caution is in order. The term “dataflow” is often used loosely for semantics that bear little resemblance to those outlined by Dennis in 1975 [15]. Most, however, can be described formally as special cases of *dataflow process networks* [33], which are in turn are a special case of *Kahn process networks* [23].

In Kahn process networks, a number of concurrent processes communicate by passing streams of data *tokens* through unidirectional FIFO channels, where writes to the channel are non-blocking, and reads are blocking (see figure 5). This means that writes to the channel always succeed immediately, while reads block until there is sufficient data in the channel to satisfy them. In particular, a process cannot test an input channel for the availability of data and then branch conditionally. Testing for available data constitutes a read, and will block the entire process until data is available. This restriction helps to assure that the program is *determinate*, meaning that its outputs are entirely determined by its inputs and those aspects of the program that are specified by the programmer.

In dataflow process networks, each process consists of repeated *firings* of a dataflow *actor* (see figure 6). A firing is a (often functional) quantum of computation. By dividing processes into actor firings, the considerable overhead of context switching incurred in most implementations of Kahn process networks is avoided.

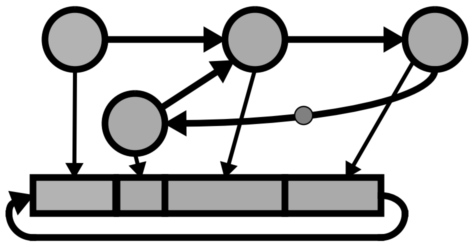


FIGURE 7. Static scheduling of a dataflow process network.

In fact, in many of the signal processing environments, a major objective is to statically (at compile time) schedule the actor firings (see figure 7). The firings are organized into a list (for one processor) or set of lists (for multiple processors). In figure 7, a dataflow graph is shown mapped into a single processor schedule. Thus, the lower part of the figure represents a list of firings that can be repeated indefinitely. A basic requirement of such a schedule is that one cycle through the schedule should return the graph to its original state (the number of tokens on each arc should be the same after the cycle as before). This is not always possible, but when it is, considerable simplification results.

Many possibilities have been explored for precise semantics of dataflow coordination languages, including for example the computation graphs of Karp and Miller [28], the synchronous dataflow graphs of Lee and Messerschmitt [31], the cyclo-static dataflow model of Lauwereins, *et al*. [29][7], the Processing Graph Method (PGM) of Kaplan, *et al.* [27], Granular Lucid [22], and others [1][14][12][40]. Many of these limit expressiveness in exchange for considerable advantages such as compile-time predictability.

Synchronous dataflow (SDF) and cyclo-static dataflow both have the particularly useful property that a finite static schedule can be quickly found that will return the graph to its original state, if such a schedule exists. This allows for extremely efficient implementations.

A key property of dataflow processes is that the computation consists of atomic firings. Within a firing, anything can happen. In many existing environments, a firing can only be specified in a host language with imperative semantics, such as C and C++. In the Ptolemy system [10], it can consist of a quantum of computation specified with any of several models of computation.

5.2 DISCRETE EVENT

As described above, the discrete-event model of computation has events with time stamps. The role of the scheduler is to keep a list of events sorted by time stamp and to process the events in chronological order. There are, however, some subtle-

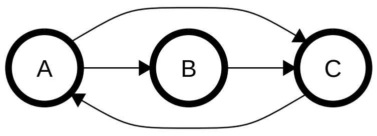


FIGURE 8. A discrete-event example.

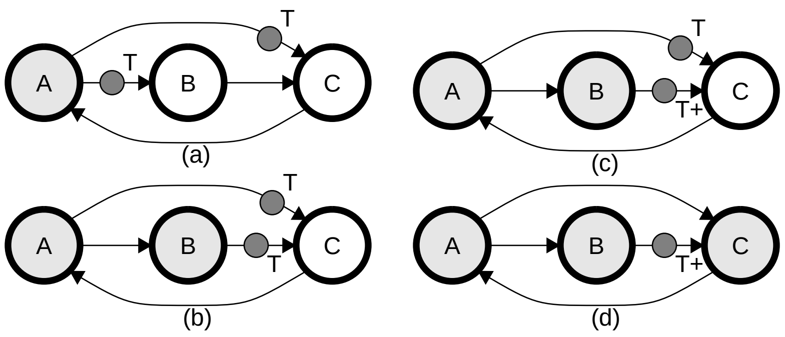


FIGURE 9. Simultaneous events in discrete-event systems.

ties that are dealt with differently in different systems. The main difficulties concern how *simultaneous events* (those with the same time stamp) are dealt with, and how *zero-delay* feedback loops are managed.

Consider the graph shown in figure 8. Suppose it specifies a program in a discrete-event coordination language. Suppose further that B is a zero-delay component. This means that each output has the same time stamp as the most recent input. Thus, if A produces one event on each of its two outputs with the same time stamp *T*, then there is an ambiguity about whether B or C should be invoked next. This situation is illustrated in figure 9(a). B and C have events at their inputs with identical time stamps, so either could be invoked next. But the behavior of C could be different in the two circumstances.

Suppose B is invoked first, resulting in the configuration shown in figure 9(b). Now, depending on the simulator, C might be invoked once, observing both input events in one invocation. Or it might be invoked twice, processing the events one at a time. In the latter case, there is no clear way to determine which event should be processed first.

Some discrete-event simulators leave this situation ambiguous. Such simulators are *nondeterminate*. In most applications, this is not desirable. A partial solution provided in some simulators is the *infinitesimal delay*. If B has an infinitesimal delay, then its output events will have time stamps that are ordered after those of the inputs even if they represent the same physical time. Then, firing A followed by B will result in the situation shown in figure 9(c), where the effect of the infinitesimal delay is indicated by the "*T*+". The next firing of C will observe only the first event, the one with time stamp *T*. This is the next one in the event queue. After this firing of C, the event with time stamp *T*+" remains to be processed, as shown in figure 9(d).

Infinitesimal delays are not an entirely satisfactory solution. Suppose the designer wishes for C to see both events at once, as in figure 9(b). There is no way to ensure that B will be invoked before C. For this reason, the discrete event domain in Ptolemy uses a different solution [10]. Graphs specifying discrete event programs are topologically sorted, and a priority is assigned to each arc. The topological sort is based on an annotation of the nodes in the graph indicating whether the node can have zero delay from any particular input to any particular output. When such zero delay is possible, the topological sort views this as a precedence constraint. Ignoring the feedback arc in figure 8, this would resolve all ambiguities. The topological sort would indicate that B should always be invoked before C when they have events at their inputs with identical time stamps. This sort of precedence analysis is identical to that done in synchronous languages (Esterel, Lustre, and Signal) to ensure that simultaneous events are processed in a deterministic way.

Of course, the feedback loop in figure 8 creates a problem. The same problem occurs in synchronous languages, where such loops are called causality loops. No precedence analysis can resolve the ambiguity. In synchronous languages, the compiler may simply fail to compile such a program. In the discrete-event domain in Ptolemy, we permit the user to annotate the arcs the graph to break the precedences. Thus, the programmer could annotate the leftward pointing arc in figure 8, again resolving the ambiguities. If the user fails to provide such annotation, a warning is issued, and the precise behavior is arbitrary (nondeterminate).

5.3 MIXING DISCRETE EVENTS AND DATAFLOW

In Ptolemy, a *domain* defines the semantics of a coordination language. But domains are modular objects that can be mixed and matched at will. Object-oriented principles are used to hide information about the semantics of one domain from another.

The discrete event (DE) domain in Ptolemy is used for time-oriented simulations of systems such as queueing networks, communication networks, and high-level computer architectures (processors, disks, caches, etc.). Many such systems contain subsystems that are better modeled in dataflow, such as signal processing subsystems. But the dataflow domains in Ptolemy have no notion of time. How can these models of computation be mixed?

Consider the case of a dataflow model inside a DE model , as shown in figure 10. In Ptolemy, the dataflow subsystem appears to the DE simulator as a zero-delay block. Suppose, for example, that an event with time stamp *T* is available at

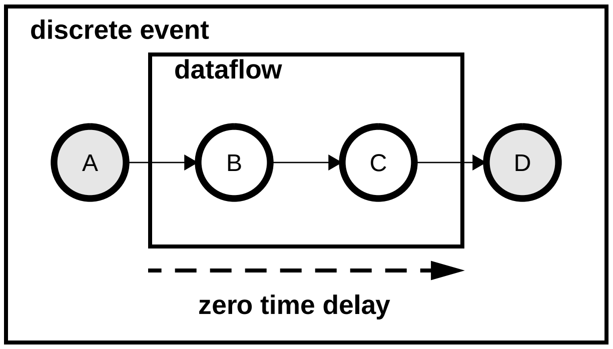


FIGURE 10. A dataflow subsystem as a module within a DE simulation.

the input to the dataflow subsystem. Then when the DE scheduler reaches this simulated time, it fires the dataflow subsystem, turing control over to the dataflow scheduler.

The question remains, how much work should the dataflow scheduler do before returning control to the DE scheduler? One possibility would be to fire a single dataflow actor, say B in figure 10, to respond to the event. But this will produce no output, and it is unclear when the dataflow scheduler should be invoked again to continue responding to the input event. Moreover, if a single actor were to be replaced by a functionally equivalent cluster of actors, the behavior of the dataflow graph would change in a fundamental way. It would take several invocations to produce the same result.

A more reasonable alternative is to fire enough dataflow actors to return the dataflow graph to its original state. Thus, if all arcs start with zero tokens, and a token arrives at the input, the scheduler should fire the actors a minimal number of times to return all arcs to zero tokens. This set of firings is called a *complete cycle* [32]; it forms a *quantum of computation*.

Consider the simplest form of dataflow, known as *homogeneous synchronous dataflow*, where all actors produce and consume exactly one token on each input or output port. For a homogeneous SDF graph, a complete cycle always consists of exactly one firing of each actor. Suppose that the dataflow graph in figure 10 is homogeneous SDF. Then when the dataflow subsystem is invoked by the DE scheduler in response to an event with time stamp *T*, actors B and C will each fire once, in that order. Actor C will produce an output event, which when passed to the DE domain will be assigned the time stamp *T*. The DE scheduler continues processing by firing actor D to respond to this event.

In the more general form of SDF, actors can produce or consume more than one token when they fire (but they always produce and consume the same number on each firing). SDF graphs always have a finite complete cycle that can be found efficiently, and it contains at least one firing of each actor in the graph. General SDF is useful for modeling multirate signal processing systems, among other applications.

If the SDF subsystem in figure 10 is a multirate system, the effects of the combined DE/SDF system are somewhat more subtle. First, a single event at the input of the subsystem may not be sufficient to cycle through one iteration of the SDF schedule. Suppose for example that actor B requires two input tokens to fire. In this case, the SDF scheduler will simply return control to the DE scheduler, having produced no output events. Only when enough input events have accumulated will any output events be produced. Secondly, when output events are produced, more than one token may be produced at a time. In Ptolemy, all such output tokens are assigned the same time stamp.

The notion of a complete cycle gets more difficult with more general dataflow models [32]. Unfortunately, for general dataflow graphs, the existence of a complete cycle is undecidable [9]. The only automatic solution we have identified is to define a quantum of computation to be a complete cycle when it exists and can be found. Otherwise, it will be implementation dependent. Alternatively, the programmer can annotate the dataflow graph to specify what firings constitute a quantum of computation. Fortunately, most signal processing algorithms have dataflow graphs for which a complete cycle exists, can be found and is finite.

Consider the reverse scenario, where a DE subsystem is included within an SDF system. The policy followed in Ptolemy is that a global notion of *current time* is maintained. For domains (such as the dataflow domains) that have no notion of time, this global time is maintained transparently. A dataflow system can be configured so that each complete cycle advances the global time by some fixed amount. This corresponds naturally to the representation of a sample rate in a signal processing system. Thus, when the outer SDF system chooses to fire the inner DE subsystem, the input events to the DE subsystem are assigned this global time as their time stamps. The inner DE scheduler is told to then process all input events up to and including events with the global current time as their time stamp. This is certainly not the only possibility, but it is unambiguous and seems to work well in practice.

A key requirement in this case is that when the DE subsystem is fired, it must produce an output event on each output port, since these will be expected by the SDF subsystem. A very simple example is shown in figure 11. The DE subsystem in the figure routes input events through a time delay (provided by the *Server* block). The events at the output of the time delay, however, will be events in the future (in simulated time). The *Sampler* block, therefore, is introduced to produce an output event at the current simulation time. This output event, therefore, is produced before the DE scheduler returns control to the output SDF scheduler, and the SDF system gets the events it expects.

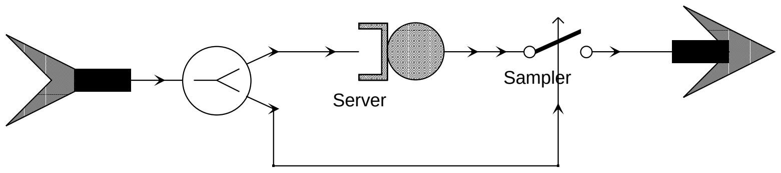


FIGURE 11. A DE subsystem designed for inclusion within an SDF system.

The behavior shown in figure 11 may not be the desired behavior. The *Sampler* block, given an event on its control input (the bottom input), copies the most recent event from its data input (the left input) to the output. If there has been no input data event, then a zero-valued event is produced. There are many alternative ways to ensure that an output event is produced. For this reason, the mechanism for ensuring that this output event is produced is not built into Ptolemy. The programmer must understand the semantics of the interacting domains, and act accordingly.

Other combinations of models of computation, including for example register-transfer-level circuit simulators and communicating processes, are discussed in the Ptolemy documentation.

5.4 HIERARCHICAL FINITE STATE MACHINES

A number of modern programming methodologies, aimed at the domain of control-dominated applications, are based on finite-state machines. Simple FSMs, however, have a major weakness; nontrivial systems have a very large number of states. Modern solutions use hierarchy, in which a single state represents an entire subsystem, and concurrency, in which multiple FSMs operate simultaneously and communicate through signals.

Statecharts and Variants. Perhaps the best known of the FSM models that support hierarchy and concurrency are the statecharts formalism [20] and at least 20 variants [41]. Most of these have a visual syntax. Commercial systems include Statemate, from iLogix [21], VisualHDL from Summit Design Inc., SpeedChart from Speed Electronics Inc., and StateVision from Vista Technologies.

Many of the differences between the variants of statecharts are in the semantics of concurrency in parallel FSMs and the communication between them. Most statecharts formalisms use the notion of instantaneous broadcast, in which input events or events produced by internal transitions are visible throughout some scope and can trigger other transitions. These other transitions in turn can produce events that trigger other transitions. Serious problems arise when the automata are modeled as reacting instantaneously, because zero-delay loops can occur, much like the zero-delay loops in discrete-event models. Unlike discrete-event models, these zero-delay loops can result in logical inconsistencies (rather than just nondeterminism). For example, a transition might produce an event that invalidates the conditions under which the transition occurred. Or a pair of transitions might produce events that trigger each other, but neither occurs because neither is enabled initially.

There are at least two fundamental interpretations of such zero-delay loops, *microsteps* and *fixed points*. In the microsteps interpretation, the actions occurring in a given time instant have a natural order. In the fixed point interpretation, they are genuinely simultaneous. In both cases, there are at least two possible outcomes of such zero-delay loops, an *instantaneous dialog* or a *contradiction*. In the former, all automata involved in the zero-delay loop end up in a well-defined state after all events at a given time have been processed. In the latter, an automaton makes a state transition at time T, issues an event that triggers a state transition in another automaton, and that other automaton issues an event that invalidates the state transition taken by the first automaton. This is one form of a *causality loop*, and is usually considered an error in the program. Unfortunately, it is not always possible for a compiler to detect such causality loops.

Most of the variants of statecharts are members of the class of *synchronous languages*. Fundamentally, this means that a program in the language fully defines the order of events. Any two events are either unambiguously simultaneous, or one precedes the other. This makes it possible to have a global *tick* that determines when automata can change state. All automata with enabled transitions change state simultaneously in response to events. The key problem is that the enabling conditions for a transition may not be testable before the tick has been at least partially processed. A simple but highly restrictive solution is to require any transition to produce events that will only be visible in the next tick. In a more common solution, causality analysis in the compiler determines when enabling conditions are testable.

Synchronous languages, with their clear notion of a tick, mix easily with other models of computation. An atomic invocation of a subsystem in such a language consists of processing one tick.

Esterel. A language with a textual syntax that describes synchronous automata particularly elegantly is Esterel [8]. An example of an Esterel program and the corresponding FSM is shown in figure 12. This program has three inputs, A, B, and R (the latter for “Reset”) and one output O. Its behavior is that after starting, or after a reset signal, it waits until it has received events on both A and B. Once that has occurred, it emits O.

Esterel has familiar imperative semantics. Two statements separated by a semicolon, as in “S1; S2”, will execute in sequence. If S1 has no code that waits

```
module EsTest:
input A, B, R;
output O;
loop
    do
        [await A || await B];
        emit O;
        halt
    watching R
end loop
end module
```

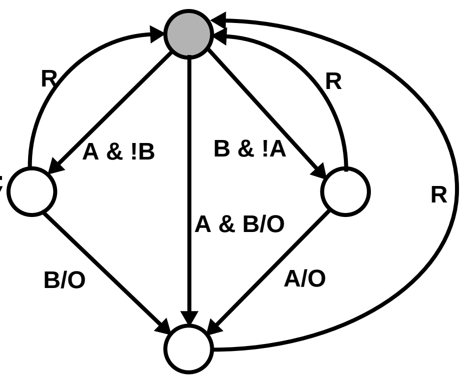


FIGURE 12. An Esterel program and its corresponding FSM.

for events, then S1 and S2 execute logically at the same instant (in the same tick). Two statements separated by a double bar, as in "S1 || S2", execute in parallel. The composite statement "[S1 || S2]" executes until *both* S1 and S2 terminate, and then it terminates. Thus, in "[S1 || S2]; S3", S3 will be executed only after both S1 and S2 have executed and finished. Thus, the statement "[await A || await B]; emit O" will emit the output in the first tick when both A and B have occurred.

The "do ... watching R" construct in figure 12 is typical of Esterel programs, and illustrates one of its major strengths. The body is executed until either it terminates or an event occurs on the signal R. Thus, this construct provides an interrupt mechanism in an intuitive way.

Comparing the Esterel program and the FSM in figure 12, we observe that if the number of signals to watch (which is two, A and B, in figure 12) increases, the size of the Esterel program will increase linearly, while the size of the FSM will increase exponentially. This illustrates a traditional problem with FSMs, *state explosion*.

Statecharts were developed in part to deal with the same problem. The same program is shown in a statecharts representation in figure 13. Two concurrent FSMs monitor the signals A and B. When both FSMs transition to their final state, then the FSM one level up in the hierarchy transitions to its "done" state. If a reset signal R is received at any time, the self-loop at the highest level of the hierarchy is triggered, reinitializing all FSMs. Like the Esterel program, the size of this program grows linearly with the number of signals that are being monitored.

5.5 MIXING CONTROL WITH DATAFLOW OR DISCRETE EVENT

Real-time embedded systems frequently combine intensive numerical computations with control. A wireless modem, for example, contains sophisticated sig-

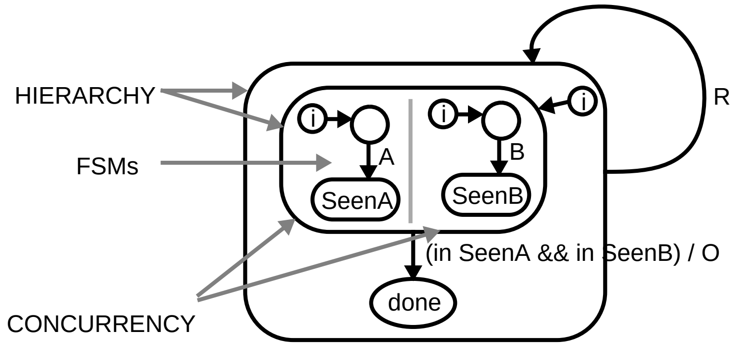


FIGURE 13. Statecharts representation of the same program as in figure 12.

nal processing including adaptive filters, phase-locked loops, encoders and decoders; but a sizable portion of the development effort for such a modem is in the software that manages the initiation of a connection, the configuration of the device, and the interaction with the host computer and the user. Modern adaptive signal processing algorithms also require much more sophisticated control than classical adaptive algorithms, often including knowledge-based decision making. Multimedia services and telecommunication services inevitably require sophisticated control, because of the key role of user interaction, in addition to sophisticated signal processing.

Mixing either hierarchical FSMs or Esterel with discrete-event models is simple and intuitive. The notion of a “tick” makes it very clear what a firing of a control subsystem means. The quantum of computation is a single tick. Moreover, these control models, like DE, react to the presence or absence of events.

Mixing these control-oriented models with synchronous dataflow is also simple. When events cross the boundary, the absence of events in the control-oriented domain must be indicated by special null tokens, which might be quite inefficient. Avoiding this inefficiency by using more general dynamic dataflow models remains a research problem.

5.6 DIGITAL HYBRID SYSTEMS

The term *hybrid systems* is used in the control theory community to refer to continuous-time systems that interact with discrete-event controllers. The continuous-time systems are typically described using differential equations. The controllers are usually specified using languages with finite-state machine semantics, in which case the model is called “hybrid automata” [2]. In our case, instead of a continuous-time plant, we have a discrete-time system specified using dataflow

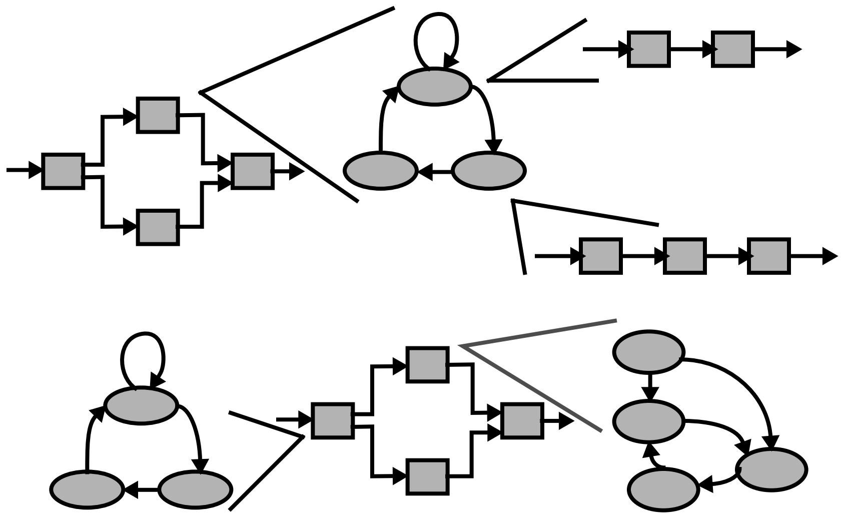


FIGURE 14. Hierarchical nesting of FSM controllers with dataflow graphs.

graphs. Instead of differential equations, these discrete-time subsystems will often be specifiable using difference equations, which in turn are specified using dataflow. We call this variant of the model *digital hybrid systems* since it preserves the essential features of the classical hybrid systems but is better suited to specification and analysis of digital systems.

Our version of digital hybrid systems consists of a hierarchical nesting of dataflow graphs with FSMs, as shown in figure 14. The depth and ordering of the nesting is arbitrary. In that figure, we have schematically illustrated dataflow semantics with rectangular boxes, and FSM semantics with round nodes. The idea is that any dataflow actor can have its functionality specified by an FSM. The FSM in turn can have actions associated with either states or transitions, and these actions can be specified by a dataflow graph. Either model can form the top level of a design.

Our implementation strategy is depicted in figure 15. A dataflow actor (labeled “dataflow wormhole”) has associated with it an FSM and a number of Ptolemy wormholes in arbitrary domains. A wormhole in Ptolemy is a subsystem with a foreign model of computation, treated as a black box. The inner wormholes contain the implementations of actions. The FSM selects among the actions. The inputs to the outer wormhole are used by either the FSM (to trigger state transitions), by the inner wormholes (probably for numerical computation), or by both.

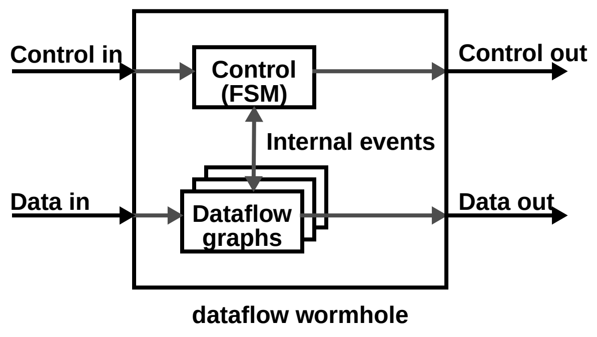


FIGURE 15. Generalization of the Ptolemy wormhole abstraction to support embedded controllers. The dataflow graphs could be replaced by any model.

5.7 MODULAR SEMANTICS

The above examples of combinations suggest that mixing models of computation is reasonably straightforward. In fact, these examples have been carefully chosen to avoid numerous treacherous pitfalls. A key problem is to determine what it means to execute a quantum of computation in a given model of computation.

When any of the above models of computation are nested within another, the outer domain invokes a quantum of computation in the inner domain. For this to have meaning, the inner domain must have a clear notion of quantum of computation. For ordinary finite-state machines, which are sequential, it is clear; a quantum of computation consists of a single state transition. But what about concurrent state machines? Should a quantum of computation be a single state transition in each state machine? What if the state machines run asynchronously, and a state transition in one only gets triggered by events in another? What if they are synchronous, but operate at different rates? The answer seems to depend on the model used to manage the concurrency and communication in the FSMs. Indeed these models are typically either dataflow, synchronous (in the sense of synchronous languages), cycle-based, or discrete-event. Thus, hierarchical FSMs that support concurrency can be viewed as manifestations of precisely the sort of heterogeneity we are discussing in this chapter.

Figure 13 identifies three orthogonal semantic properties of statecharts, FSMs, hierarchy, and concurrency. Although it is not possible with statecharts (because of transitions that cross hierarchical boundaries), a simpler model would separate the semantics of concurrency from the semantics of FSMs. Thus, a model only slightly weaker than statecharts would be, in fact, a composition of two models, much like the dataflow/FSM combination shown in figure 14. Since simple

FSMs have a clear and unambiguous notion of a quantum of computation (one state transition), the problem reduces to determining what a quantum of computation is in the semantic model used to manage the concurrency and communication. This model could be a synchronous model, a dataflow model, or a discrete-event model, yielding three variants of the statecharts idea. But unlike the monolithic variants described in [41], these variants are modular. They are created conveniently when needed simply by choosing the appropriate model of computation at each level of the hierarchy.

6. Acknowledgments

We would like to thank the Ptolemy team for building a magnificent laboratory for experimenting with the concepts discussed in this chapter. The Ptolemy project is supported by the Advanced Research Projects Agency and the U.S. Air Force (under the RASSP program, contract F33615-93-C-1317), the Semiconductor Research Corporation (SRC) (project 95-DC-324-016), the National Science Foundation (MIP-9201605), the State of California MICRO program, and the following companies: Bell Northern Research, Cadence, Dolby, Hitachi, Mentor Graphics, Mitsubishi, Motorola, NEC, Pacific Bell, Philips, and Rockwell.

7. References

1. W. B. Ackerman, “Data Flow Languages,” *Computer,* Vol. 15, No. 2, pp 15-25, February 1982.
2. R. Alur, C. Courcoubetis, T. A. Henzinger, P.-H. Ho, “Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems,” *LNCS 736*, Springer-Verlag, Berlin, 1993.
3. A. Benveniste and G. Berry, “The Synchronous Approach to Reactive and Real-Time Systems,” *Proceedings of the IEEE*, Vol. 79, No. 9, 1991, pp. 1270-1282.
4. A. Benveniste and P. Le Guernic, “Hybrid Dynamical Systems Theory and the SIGNAL Language,” *IEEE Tr. on Automatic Control*, Vol. 35, No. 5, pp. 525-546, May 1990.
5. J. Bier, P. Lapsley, E. A. Lee, and F. Weller, “DSP Design Tools and Methodologies,” *Technical Report*, Berkeley Design Technology, 39355 California St., Suite 206, Fremont, CA 94538, 1995.
6. J. Bier, P. Lapsley, and E. A. Lee, “Buyer’s Guide to DSP Processors,” *Technical Report*, Berkeley Design Technology, 39355 California St., Suite 206, Fremont, CA 94538, 1994.
7. G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, “Static Scheduling of Multi-rate and Cyclo-Static DSP Applications”, *Proc. 1994 Workshop on VLSI Signal Processing*, IEEE Press, 1994.
8. F. Boussinot, R. De Simone, “The ESTEREL Language,” *Proceedings of the IEEE,* Vol. 79, No. 9, pp 1293-1304, September 1991.
9. J. T. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model,* Tech. Report UCB/ERL 93/69, Ph. D. Dissertation, Dept. of EECS, University of California, Berkeley, CA 94720, 1993.
10. J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, “Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems,” *Int. Journal of Computer Simulation*, special issue

on “Simulation Software Development,” vol. 4, pp. 155-182, April, 1994. (http://ptolemy.eecs.berkeley.edu/papers/JEurSim.ps.Z).

11. B. Cogswell and Z. Segall, “Timing Insensitive Binary to Binary Translation of Real Time Systems,” *Technical Report*, ECE Dept., Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213.
12. F. Commoner and A. W. Holt, “Marked Directed Graphs,” *Journal of Computer and System Sciences*, Vol. 5, pp. 511-523, 1971.
13. R. Camposano, “From Behavior to Structure: Highlevel Synthesis”, *IEEE Design and Test of Computers*, Oct. 1990, pp 8-19.
14. N. Carriero and D. Gelernter, “Linda in Context,” *Comm. of the ACM*, Vol. 32, No. 4, pp. 444-458, April 1989.
15. J.B. Dennis, “First Version Data Flow Procedure Language”, Technical Memo MAC TM61, May, 1975, MIT Laboratory for Computer Science.
16. H. De Man, F. Catthoor, G. Goossens, J. Vanhoof, J. Van Meerbergen, S. Note, J. Huisken, “Architecture-driven synthesis techniques for mapping digital signal processing algorithms into silicon,” *Proceedings of the IEEE*, Vol. 78, No. 2, pp. 319-335, February, 1990.
17. B. L. Evans, S. X. Gu, A. Kalavade, and E. A. Lee, “Symbolic Computation in System Simulation and Design,” *Proc. of SPIE Int. Sym. on Advanced Signal Processing Algorithms, Architectures, and Implementations*, July 9-16, 1995, San Diego, CA.
18. G. Goosens, F. Cathoor, D. Lanneer, and H. De Man, “Integration of Signal Processing Systems on heterogeneous IC architectures”, Proceedings of the Sixth International Workshop on High-Level Synthesis, Laguna Niguel, CA, Nov. 1992.
19. N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, “The Synchronous Data Flow Programming Language LUSTRE,” *Proceedings of the IEEE*, Vol. 79, No. 9, 1991, pp. 1305-1319.
20. D. Harel, “Statecharts: A Visual Formalism for Complex Systems,” *Sci. Comput. Program.*, vol 8, pp. 231-274, 1987.
21. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, M. Trakhtenbrot, “STATEMATE: A Working Environment for the Development of Complex Reactive Systems,” *IEEE Tr. on Software Engineering*, Vol. 16, No. 4, April 1990.
22. R. Jagannathan, “Parallel Execution of GLU Programs,” presented at *2nd International Workshop on Dataflow Computing*, Hamilton Island, Queensland, Australia, May 1992.
23. G. Kahn, “The Semantics of a Simple Language for Parallel Programming,” *Proc. of the IFIP Congress 74*, North-Holland Publishing Co., 1974.
24. A. Kalavade and E. A. Lee, “A Hardware/Software Codesign Methodology for DSP Applications,” *IEEE Design and Test of Computers*, September 1993, vol. 10, no. 3, pp. 16-28.
25. A. Kalavade and E. A. Lee, “The Extended Partitioning Problem: Hardware/Software Mapping and Implementation-Bin Selection,” *Proc. of IEEE Int. Workshop on Rapid Systems Prototyping*, Chapel Hill, NC, June, 1995 (http://ptolemy.eecs.berkeley.edu/papers/extended_partitioning/).
26. A. Kalavade and E. A. Lee, “Manifestations of Heterogeneity in Hardware/Software Codesign,” *Proc. of Design Automation Conference*, San Diego, CA, June, 1994, pp. 437-438 (http://ptolemy.eecs.berkeley.edu/papers/codesign).
27. D. J. Kaplan, *et al.*, “Processing Graph Method Specification Version 1.0,” Unpublished Memorandum, The Naval Research Laboratory, Washington D.C., December 11, 1987.
28. R. M. Karp, R. E. Miller, “Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing,” *SIAM Journal*, Vol. 14, pp. 1390-1411, November, 1966.
29. R. Lauwereins, P. Wauters, M. Adé, J. A. Peperstraete, “Geometric Parallelism and Cyclo-Static Dataflow in GRAPE-II”, *Proc. 5th Int. Workshop on Rapid System Prototyping*, Grenoble, France, June, 1994.

30. J. C. Lee, E. Cheval, and J. Gergen, “The Motorola 16-Bit DSP ASIC Core”, Proceedings of ICASSP 1990, V3.12, pp 973-976.
31. E. A. Lee and D. G. Messerschmitt, “Synchronous Data Flow,” *IEEE Proceedings*, September, 1987.
32. E. A. Lee, “Consistency in Dataflow Graphs”, *IEEE Transactions on Parallel and Distributed Systems”*, Vol. 2, No. 2, April 1991.
33. E. A. Lee and T. M. Parks, “Dataflow Process Networks,” *Proceedings of the IEEE*, May 1995. (http://ptolemy.eecs.berkeley.edu/papers/processNets)
34. P. G. Paulin, C. Liem, T. C. May, S. Sutarwala, “DSP Design Tool Requirements for the Nineties: An Industrial Perspective”, *Journal of VLSI Signal Processing*, special issue on “Synthesis for DSP”, Jan. 1995, vol.9, (no.1-2) p. 23-47.
35. J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck, “Software Synthesis for DSP Using Ptolemy,” *Journal on VLSI Signal Processing*, vol. 9, no. 1, pp. 7-21, Jan., 1995 (http://ptolemy.eecs.berkeley.edu/papers/jvsp_codegen/jvsp_codegen.ps.Z).
36. J. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, “Fast Protoyping of Datapath-Intensive Architectures,” *IEEE Design and Test of Computers*, pp. 40-51, June 1991.
37. J. Rasure and C. S. Williams, “An Integrated Visual Language and Software Development Environment”, *Journal of Visual Languages and Computing*, Vol 2, pp 217-246, 1991.
38. J. Rowson, “Hardware/software Co-simulation”, *Proc. of the 31st Design Automation Conference*, San Diego, June, 1994, pp 439-40.
39. S. Ritz, M. Pankert, and H. Meyr, “High Level Software Synthesis for Signal Processing Systems,” in *Proc. of the Int. Conf. on Application Specific Array Processors*, IEEE Computer Society Press, August 1992.
40. P. A. Suhler, J. Biswas, K. M. Korner, J. C. Browne, “TDFL: A Task-Level Dataflow Language”, *J. on Parallel and Distributed Systems,* 9(2), June 1990.
41. M. von der Beeck, “A Comparison of Statecharts Variants,” in Proc. of Formal Techniques in Real Time and Fault Tolerant Systems, LNCS 863, pp. 128-148, Sprinter-Verlag, Berlin, 1994.
42. P. Zepter and T. Grötker, “Abstract Multirate Dynamic Data-Flow Graph Specification for High Throughput Communication Link ASICs”, *IEEE VLSI DSP Workshop*, The Netherlands, 1993.