RAISING THE LEVEL OF ABSTRACTION: A SIGNAL PROCESSING SYSTEM DESIGN COURSE

Brian L. Evans and Güner Arslan

Dept. of Electrical and Computer Engineering,
The University of Texas at Austin, Austin, TX, 78712-1084 USA
E-mail: {bevans,arslan}@ece.utexas.edu

ABSTRACT

Signal processing systems are inherently heterogeneous. They often contain a mixture of filtering, communication, and control algorithms implemented by a variety of technologies such as digital hardware, software, and analog circuits. Modern system-level design manages heterogeneity by first decoupling specification from implementation and then decomposing the specification into a hierarchy of simpler block diagrams. Each block diagram can be associated with a different set of formal rules governing its behavior, so the hierarchical composition becomes heterogeneous. The same system specification can be executed for simulation or synthesized into a variety of implementations. In this paper, we describe a graduate-level signal processing system design course that presents this modern approach. The course covers block diagram modeling, algorithm specification, system simulation, and system synthesis. Students gain hands-on experience by modifying university and commercial system-level CAD tools. This paper proposes steps to transition this course into the undergraduate curriculum.

1. INTRODUCTION

Signal processing systems implement multiple styles of algorithms (e.g. filtering, modulation/demodulation, and feedback control) using a variety of technologies (e.g. digital signal processors, field-programmable gate arrays, microcontrollers, application-specific integrated circuits, and operating systems). Teaching signal processing systems often reduces to courses that target one style of algorithm and/or one implementation technology. For example, students in a real-time digital signal processing laboratory might develop software for a single digital signal processor to implement a real-time system, e.g. a voiceband modem [1]. Students in a digital VLSI course would either learn transistor-level layout using Magic or synthesis of designs using tools from Synopsys, Mentor Graphics, or Cadence.

Modern system-level design manages heterogeneity by first decoupling specification from implementation and then decomposing the specification into a hierarchy of simpler block diagrams, as shown in Fig. 1. Each block diagram can be associated with a different set of formal rules, called a model of computation, to govern its behavior. Models of computation that could be used image and video processing systems are listed in Table 1. Ideally, the same system specification could be executed for simulation or synthesized into a variety of implementations.

This research was supported by a US National Science Foundation CAREER Award under grant MIP-9702707 and the US Defense Advanced Research Projects Agency under DARPA Grant DAAB07-97-C-J007.

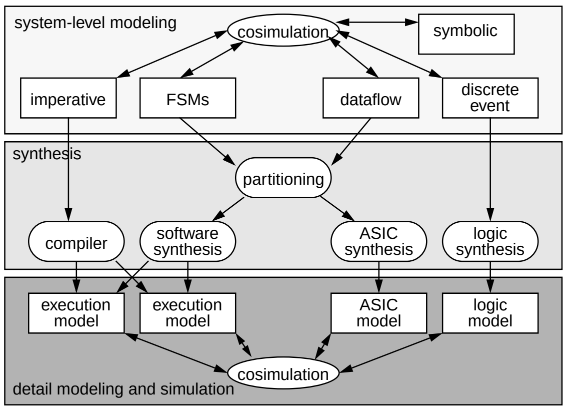

Figure 1: Heterogeneity in the top-down design flow of complex systems. Drawing is copyright © 1994 by Edward A. Lee. Used by permission.

We introduce a new graduate course to present modern techniques for the design and implementation of signal processing systems. The course raises the level of abstraction to specifying, simulating, and synthesizing entire signal processing systems. In the course, the student

- learns implementation-unbiased models of computation,
- composes these models to specify complex systems
- simulates complex heterogeneous systems, and
- synthesizes systems onto hardware/software technologies.

Students gain hands-on experience by modifying two system-level design environments: Ptolemy from the University of California at Berkeley and the Advanced Design System from HP EEsof. All of the notes, homework assignments, review material, and past student project reports for the class are on the Web at

http://www.ece.utexas.edu/~bevans/courses/ee382c/

Subsystem	Model of Computation
audio processing	1-D dataflow
digital image processing	2-D dataflow
image/video resampling	m-D multirate dataflow
user interface	synchronous/reactive
communication protocols	finite-state machine
digital control	dataflow
image understanding	knowledge-based control
scalable descriptions	process networks

Table 1: Models of computation for describing the signal processing, communications, and control aspects of image and video processing systems.

Table 2 lists the lectures in the course. Section 2 describes the first six lectures on system-level performance, design, and specification. Section 3 discusses models of computation. Section 4 describes algorithms for scheduling models of computation for simulation and synthesis. Section 5 overviews the system-level design environments used in the course. Section 6 proposes steps to transition this new graduate course into the undergraduate curriculum. Section 7 concludes the paper.

2. INTRODUCTORY LECTURES

The first lecture, "System Performance Measures," defines an embedded system as the "part of a product with which the end user does not directly interact or control." Constraints on the design and implementation of embedded systems often include weight, volume, power consumption, and economic cost. Concerning power consumption, the course surveys the performance of of the four major battery families in embedded systems (NiCd, NiMH, Li^+, and Zn Air). It also gives the power consumption of two leading processors— Intel's Pentium with MMX and the Texas Instruments TMS320C62x VLIW Digital Signal Processor (DSP)— to show how power hungry those processors are. The first lecture points out that it takes about 1 kg of Lithium Ion batteries to power an 266 MHz Pentium Processor for 22 hours or a 120 MHz TMS320C62x processor for 96 hours. Yet, both processors post nearly identical benchmarks on digital signal processing algorithms. This raises the issue that clock speed and MIPS are not particularly meaningful measures of performance.

The "System Level Design" introductory lecture describes system-level design s a way to coordinate the execution of and communication between subsystems. System-level design concerns cosimulation of a system specification that may be a mixture of hardware components, software components, and algorithms; cosynthesis of a system specification onto a specific hardware and software architecture, possibly including the generation of operating systems; and codesign of a hardware and software architecture best suited for a class of systems (e.g. video compression). System-level design seeks to meet global system-level constraints on throughput and delay, while possibly minimizing area and power: these are global optimizations. Next, these global optimizations are contrasted with the local optimization performed by compilers for software or manually in cell designs in VLSI libraries. Optimization requires the formation of a cost function which in turn relies on measures. Measures can be estimates of complexity or results of an implementation generated by another CAD tool.

Topic	Lectures
Introduction	System Performance Measures
	System-Level Design
	Digital Signal Processors †
	Block Diagram Languages I
	Block Diagram Languages II
	Block Diagram Languages III
Models of Computation ‡	Dataflow Modeling
	Synchronous Dataflow (SDF)
	Boolean Dataflow
	Dynamic Dataflow
	Process Networks
	Timed SDF
	Multidimensional SDF
	Cyclo-Static Dataflow
	Discrete Event
	Synchronous/Reactive
	Finite State Machines (FSM)
Scheduling Algorithms	Introduction to Graph Theory
	Introduction to SDF Scheduling
	SDF Looped Scheduling
	Multiprocessor SDF Scheduling
	Synthesis of FSMs
Composition of Models	Hybrid FSMs
	Mixing FSMs and Dataflow Models
Standalone Topics	Native Signal Processing
	Hardware/Software Codesign

† Also introduces 1-D interpolation and decimation.
‡ Discusses simulation and synthesis of each model.

Table 2: Each lecture is 75 minutes long. Five additional lecture periods are used for the two midterms and student presentations.

The third lecture covers the architecture of traditional digital signal processors. Traditional DSP processors, such as the Texas Instruments TMS320C54x, the Analog Devices SHARC, and the Motorola 56xxx families, have an equally small amount of on-chip program and data memory. Hence, when synthesizing code for these processors, a CAD tool should generate software that uses a minimum but equal amount of program and data memory. This turns out to be an NP-complete problem, as addressed in Section 4.

The next three lectures concern the use of block diagrams in engineering— circuit schematics, computer architecture, control theory, signal processing, and communication. Block diagrams enable a designer to divide large complex designs into smaller simpler designs using a visual syntax. The interaction between blocks in any subsystem is determined by a model of computation. Several models of computation are described in more detail in the subsequent lectures.

3. MODELS OF COMPUTATION

This course discusses dataflow, discrete event, synchronous, reactive, process networks, and finite state machine models of computation. This class presents each model in detail including its strengths, weaknesses, applications, and mathematical basis. The models of computation discussed in this class, however, do not make any assumptions about the linearity, time-invariance, or memory usage of composite blocks in the block diagram.

3.1. Dataflow

In the dataflow model, a signal is a sequence of tokens and an actor maps input tokens onto output tokens. A set of firing rules specify when an actor can fire which means a consumption of an input token and a production of an output token. A sequence of firings is called a Dataflow Process. The strengths of this model are the following:

- well-suited for data-intensive (signal processing) algorithms,
- loose synchronization,
- determinate under simple conditions, and
- maps well to hardware and software.

The model has one weakness:

- inappropriate for control-intensive systems.

Many flavors of dataflow graphs exist. The course discusses the following three: Synchronous, Boolean, and Dynamic.

3.1.1. Synchronous Dataflow

In Synchronous Dataflow (SDF), all computation and communication can be scheduled statically. An SDF graph can always be implemented in finite time using finite memory. Thus, an SDF graph can be executed over and over again in a periodic fashion without requiring additional resources as it runs. This type of operation is well-suited to digital signal processing and communications systems which often process an endless supply of data.

An SDF graph consists of nodes and arcs. Nodes represent operations which are called actors. Arcs represent data values called tokens which stored in first-in first-out (FIFO) queues. The word token is used because each data values can represent any data type (e.g. integer or real) or any data structure (e.g. matrix or image).

SDF graphs obey the following rules:

1. An actor is enabled for execution when enough tokens are available at all of the inputs.
2. When an actor executes, it always produces and consumes the same fixed amount of tokens.
3. The flow of data through the graph may not depend on values of the data.

Because of the second rule, the data that an actor consumes is removed from the buffers on the input arcs and not restored. The consequence of the last rule is that an SDF graph may not contain data-dependent switch statements such as an if-then-else construct and data-dependent iterations such as a for-loop. However, the actors may contain these constructs because the scheduling of an SDF graph is independent of the what tasks the actors do.

3.1.2. Boolean Dataflow

Unlike Synchronous Dataflow, Boolean Dataflow allows conditional flow of data in a graph. Boolean Dataflow is essentially Synchronous Dataflow with addition of conditional switch (demultiplex) and select (multiplex) actors. Boolean Dataflow is Turing equivalent, so not every Boolean Dataflow graph can be executed in finite time using finite memory. Heuristics can be used to cluster some Boolean Dataflow graphs into clusters whose input/output behavior obey Synchronous Dataflow semantics. Clusters are statically scheduled much like Synchronous Dataflow graphs, and unclustered actors are dynamically scheduled.

3.1.3. Dynamic Dataflow

Dynamic Dataflow is a combination of Synchronous and Boolean Dataflow plus its own actors which have a wait port. Enough data must be present on the wait port to be enabled. The minimum amount can change from firing to firing. Other ports obey Synchronous Dataflow semantics. The Boolean input for the Switch and Select Boolean Dataflow actors is a wait port of one token. Dynamic Dataflow requires run-time scheduling.

3.2. Discrete Event Models

In discrete event systems, changes in system behavior are marked by discrete occurrences or events, e.g. VHDL, Spice, OpNet, Bones, SimuLink, and the Discrete Event (DE) domain in Ptolemy. Events consist of a token and a time stamp. A token can be a scalar value, a matrix, a data structure, and so forth. The time stamp might be a single floating-point number (Spice, Ptolemy DE domain) or consist of an ordered pair of integers (VHDL). The scheduler runs the simulation by maintaining a sorted record of all of the time stamps of the events in the system, and advances time to the next time stamp. Thus, there is a total ordering between all events— given any two events, we can tell if they occur simultaneously or after one another.

The firing rule for when a block is active is that (new) data must be present on at least one of the inputs. According to these rules, source blocks are never fired. Instead, source blocks must put the events they produce in the global record of all of the time stamps maintained by the scheduler. This will be the first of many cases in which discrete event blocks must interact directly with the scheduler— unlike Synchronous Dataflow, the blocks and the scheduler are no longer independent.

The strengths of the Discrete Event model are the following:

- Natural description of asynchronous digital hardware,
- Global synchronization,
- Determinate under simple conditions, and
- May be simulated under simple conditions.

The weaknesses are the following:

- Expensive to implement in software, and
- May over-specify or over-model systems

3.3. Synchronous/Reactive Models

The objective of a synchronous/reactive model is to aid in designing real-time embedded controllers using concurrently-executing communicating blocks. Stephen A. Edward adopts a heterogeneous approach that allows the blocks to be specified in any language, provided their interface conforms to the model [2]. This allows subsystems to be written in the most suitable language, simplifying the designer's task. Reactive systems must run at the speed of their environment, and when something happens in these systems is often as important as what happens. For this reason, the synchronous approach has been developed, allowing control over system timing to be as precise as control over system function. This approach relies on the synchrony hypothesis, which assumes a system runs infinitely fast. This breaks time into a sequence of discrete instants and provides global synchronization.

Stephen A. Edwards presents a new system description scheme that combines synchrony with heterogeneity [2]. He formally present the semantics, which are complicated by the possibility of zero-delay feedback loops, and present an efficient, predictable execution scheme based on chaotic iteration toward a least-fixed point solution, along with results that show it practical for medium-sized examples.

The major strengths of this model are the following:

- Appropriate for control intensive systems,
- Tightly synchronized,
- Determinate in most cases, and
- Maps well to hardware and software.

The weaknesses are the following:

- Overspecifies computationally-intensive systems,
- Compromises modularity,
- Causality loops are possible, and
- Causality loops are hard to detect.

3.4. Process Networks

Process networks is a model of computation in which multiple parallel processes can execute simultaneously. The model uses a directed graph notation, where each node represents a process and each edge represents a one-way FIFO queue of data words. A producer node inserts data into the queue, while a consumer node removes them. This model is natural for describing the streams of data samples in a signal processing system. Consumers are blocked when they attempt to get data from an empty input channel. However, queues are of infinite length, so producers are not blocked. This can cause unbounded accumulation of data on a given queue. This model is determinate: the results of the computation (the data produced on the queues) does not depend on the firing order of the processes. The problems of determining whether a process network will terminate, or can be scheduled with bounded memory are undecidable.

The major strengths of this model can be listed as

- Maps easily to threads, but much easier to use
- Loose synchronization
- It is determinate under simple conditions
- Implementable under simple conditions
- It is Turing complete in the sense of being expressive

and the weaknesses are

- Control intensive systems are hard to specify
- Deadlock and bounded memory are undecidable

3.5. Finite State Machines

Finite state machines are behavioral view of sequential circuits. They describe the transitional behavior of these circuits. Finite state machines have two major representations, State Transition Graphs and Tables. Both representations are very similar, they show the output and the new state for every possible input and previous state. Specifying Finite State Machines (FSMs) is important in formalizing the design of network protocols and embedded controllers. Traditional FSMs are very close in abstraction to the implementation, and suffer from many weaknesses:

- An exponential explosion in the number of states when composing substates;
- Difficult to modify complex designs since hierarchy is not present;
- Does not easily handle global control signals such as reset and halt; and
- Computation occurs in a single path, so there is no support for concurrency.

The goal is to find models of computation that overcome these drawbacks. Hierarchy is key for managing design complexity and preventing a state space explosion when composing substates.

Models of computation for FSMs may have graphical and textual syntax. Some finite state machines are easier to describe textually and some are easier to describe graphically. Graphical specification languages include Statecharts and Argos, and textual programming languages include Esterel. Statecharts, Argos, and Esterel support hierarchy, concurrency, and various communication models. All three frameworks are used to specify reactive systems (systems that respond to the environment at the speed of the environment). All three are based on the synchrony hypothesis (all communication and computation is instantaneous).

4. SCHEDULING ALGORITHMS

After spending a lecture on introductory material of Graph Theory the scheduling problem is introduced. The firing rules for dataflow graphs, synchronous/reactive systems, and finite state machines impose partial ordering constraints on the actor firings. Scheduling algorithms constrain the partial ordering in order to meet the following practical objectives:

- scheduling cost: Scheduling decisions should be made as much as possible at compile time.
- bounded memory: The total number of unconsumed tokens should be bounded throughout the execution if this is possible for the given graph.

At the same time, we want to avoid artificial deadlock:

- deadlock: The graph should not halt if there are enabled tasks.

For many models of computation, the scheduling problem is NP-complete— no algorithm can schedule a graph in polynomial time in the size of the graph. One exception is that the scheduling for the Synchronous/Reactive model can be performed in quadratic time.

For SDF, every valid SDF graph can be executed for infinite time in bounded memory, and all scheduling decisions can be made at compile time. Compile-time scheduling is NP-complete. That is, there is no known algorithm that can schedule every possible SDF graph in polynomial time in the size of the graph (number of functional nodes plus the number of arcs connecting the nodes). In fact, class-S uniprocessor schedulers and multiprocessor schedulers that first have to convert the SDF graph into a directed acyclic graph (DAG) of precedences require exponential time in the worst case. This course discusses two heuristics for optimizing uniprocessor schedules and one heuristic for optimizing multiprocessor schedules. The uniprocessor heuristic will always find the optimal schedule for a large subset of SDF graphs. The multiprocessor heuristic clusters an SDF graph into a two-level hierarchy, schedules each child onto one processor using one of the two uniprocessor heuristics, and schedules the parent using a traditional multiprocessor scheduler based on DAG.

5. SOFTWARE TOOLS

The students use two system-level CAD tools: Ptolemy from UC Berkeley and the Advanced Design System from HP EEsof. In 1990, Ptolemy was initiated as a combination of the Synchronous Dataflow and Discrete Event models of computation. More models of computation can be added to Ptolemy by simply defining how the model passes data and control to a universal interface. Hence, the interaction between every possible pair of models does not have to be defined. Ptolemy 0.7.1 can cosimulate 11 models of computation, including those described above. Ptolemy can synthesize Synchronous Dataflow graphs onto multiple technologies, but it is relatively weak at synthesizing complex systems onto multiple technologies. Students use Ptolemy to explore the seven models of computation listed above and their interaction. They also use Ptolemy to synthesize C code, Motorola 56000 DSP assembly code, and multiple styles of VHDL, as well as mixed implementations, all from Synchronous Dataflow Graphs.

6. TRANSITION TO THE UNDERGRADUATE LEVEL

6.1. Pre-Requisites, Grading, and Textbooks

The course is currently geared for a first-year graduate students who has taken a class in signals and systems and data structures and algorithms, knows object-oriented programming, and has experience with either embedded hardware or software. An embedded software course might develop applications for microcontrollers and digital signal processors. An embedded hardware course might design digital or analog ICs. Neither a digital signal processing course nor a real-time systems course is a pre-requisite.

In the course, students analyze material by completing homework assignments and studying for tests. They synthesize material by completing a literature survey and a computer implementation, each worth 25% of the final grade. Students give presentation for both the literature survey and final report. The two midterms are each worth 20%, and homework is 10%. There is no final exam. The two course textbooks [3, 4] are written at the graduate level.

6.2. Making the Transition to a Senior Elective

As an undergraduate course, less time should be spent on the mathematical framework underlying the models of computation. Hence, the book Software Synthesis Using Dataflow Graphs would still be appropriate, but the other textbook Lattices and Partial Order would be too advanced as it requires a real analysis background. The 50% project would still be the catalyst for the students to gain the extra hands-on experience with system-level CAD tools, but the scope of the implementation should be reduced to an appropriate level. The pre-requisites can already be met by a first-semester senior in many Electrical and Computer Engineering curricula, including at UT Austin, UC Berkeley, and the Georgia Institute of Technology.

7. CONCLUSION

Students learn how to manage the fundamental problem facing the design of signal processing systems: heterogeneity. By decoupling the specification from the implementation, alternative implementations can be considered for the same specification. If a new implementation technology appears, then a CAD tool could be modified to synthesize the same specification into the new technology. Models of computation have a solid mathematical basis. They support heterogeneity in that they can be composed to characterize heterogeneous systems. They are general in that both hardware and software can be synthesized from them. Dataflow models receive the most attention as they are well-suited for describing data-intensive signal processing algorithms. Finite state machines and synchronous/reactive models, primarily used for control, are also covered. The students use Ptolemy from the University of California and the Advanced Design System from HP EEsof to evaluate models of computation and scheduling algorithms.

8. REFERENCES

[1] S. A. Tretter, Communication system design using DSP algorithms: with laboratory experiments for the TMS320C30. Plenum Press, 1995.

[2] S. A. Edwards, The Specification and Execution of Synchronous Reactive Systems. PhD thesis, University of California, Berkeley, 1997.

[3] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, Software Synthesis from Dataflow Graphs. Norwell, Massachusetts: Kluwer Academics Publishers, 1996.

[4] B. Davey and H. Priestley, Introduction to Lattices and Order. Cambridge, United Kingdom: Cambridge University Press, 1990.