A SIGNAL PROCESSING SYSTEM-LEVEL 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@ece.utexas.edu

ABSTRACT

We describe a first-year graduate course in embedded system design. Embedded systems may contain a mixture of signal processing, communication, and control algorithms implemented by a variety of technologies such as digital hardware, software, and analog circuits. The course focuses on how modern design methods and tools handle the heterogeneity and complexity in embedded systems design and covers block diagram modeling, algorithm specification, system simulation, and system synthesis. Students gain hands-on experience by using electronic design automation tools to design and simulate systems. This paper proposes steps to transition this course into the undergraduate curriculum.

1. INTRODUCTION

Embedded systems implement multiple styles of algorithms (e.g., filtering, modulation/demodulation, and feedback control) using a variety of technologies (e.g., digital signal processors or DSPs, field-programmable gate arrays, microcontrollers, application-specific integrated circuits, and operating systems). Teaching embedded systems, however, often reduces to targeting one style of algorithm and/or one implementation technology. For example, students in a real-time digital signal processing laboratory [1, 2, 3] might develop software for a particular DSP. A digital VLSI course might focus on transistor-level layout (e.g., using Magic [4]) or high-level synthesis (e.g., using tools by [5, 6, 7]).

Using modern embedded system design methods and tools, a designer manages heterogeneity by (1) decoupling the system specification from its implementation, and (2) decomposing the system specification into a hierarchical combination of subsystems. This decoupling allows the same system specification to be mapped onto many different target implementations, e.g., for evaluating candidate architectures, as shown in Fig. 1. Each subsystem in the system specification consists of a declaration of components and their connections (e.g., a block diagram) as well as a meaning attached to the declaration (e.g., a formal model of computation). A formal model consists of a set of mathematical rules with provable properties that govern the flow of data and control. A schedule is an ordering on the execution of the components, and a valid schedule is a schedule

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.

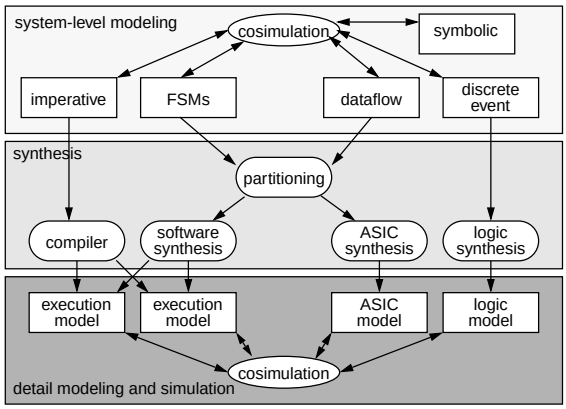

Figure 1: Heterogeneity in the top-down design flow of complex systems. Drawing is copyright © 1994 by Edward A. Lee. Used by permission.

that obeys the mathematical rules of the model of computation. Scheduling is a necessary step for either simulating or synthesizing a system. Table 1 lists models of computation that could be used in image and video processing systems.

In Spring 1997, Prof. Evans introduced a first-year graduate course, *Embedded Software Systems*, to present this modern approach [8]. The course has been taught five times to a total of 75 students. 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 Classic [9] from the University of California at Berkeley and the Advanced Design System [10] from Agilent EEsof. In Spring 2000, three textbooks [11, 12, 13] were used. Based on student suggestions to use only one textbook, the Spring 2002 offering will use only [11]. All lectures, notes, assignments, and past student projects for the class are available at

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

Table 2 lists the lectures in the course. Section 2 describes the first six lectures on system performance, design,

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.

and specification. Section 3 discusses models of computation. Section 4 describes algorithms for scheduling models of computation. 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." Design constraints for embedded systems often include weight, volume, power consumption, and economic cost. Concerning power consumption, the course surveys the performance 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 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 as 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 electronic design automation (EDA) 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 ‡	Synchronous Dataflow (SDF)
	Boolean Dataflow
	Dynamic Dataflow
	Process Networks
	Discrete Event
	Timed SDF
	Synchronous/Reactive
	Finite State Machines (FSM)
Scheduling Algorithms	Introduction to Graph Theory
	Introduction to SDF Scheduling
	SDF Looped Scheduling I
	SDF Looped Scheduling II
	Multiprocessor SDF Scheduling
Composition of Models	Hybrid FSMs
	Mixing FSMs and Dataflow Models
Standalone Topics	Native Signal Processing
	Communication Systems
	Rate Monotonic Analysis

† Also introduces 1-D interpolation and decimation.
‡ Discusses basics of simulating and scheduling 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 EDA 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, instruction set architecture, control theory, and signal processing. 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 network, and finite-state machine models of computation. This class evaluates each model, as well as gives applications and the model's mathematical basis. These particular models of computation do not make any assumptions about the linearity, time-invariance, or memory usage of composite blocks in the block diagram. The following sections summarize these models.

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 follow:

- 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 value 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 Classic. 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 follow:

- natural description of asynchronous digital hardware,
- global synchronization,
- determinate under simple conditions, and
- may be simulated under simple conditions.

The weaknesses follow:

- expensive to implement in software, and
- may over-specify or over-model systems

3.3. Synchronous/Reactive Models

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 also been developed to allow 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. A

synchronous/reactive model breaks time into a sequence of discrete instants and provides global synchronization. This model aids in the design of real-time embedded controllers using concurrently-executing communicating blocks.

A heterogeneous synchronous/reactive model would allow the blocks to be specified in any language, provided that their interface conforms to the model [14]. This allows subsystems to be written in the most suitable language, which simplifies the designer's task. Edwards' presents a heterogeneous synchronous/reactive model in [14]. He formally presents the semantics, which are complicated by the possibility of zero-delay feedback loops. He also presents an efficient, predictable execution method 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 Edward's model follow:

- appropriate for control intensive systems,
- tightly synchronized,
- determinate in most cases, and
- maps well to hardware and software.

The weaknesses follow:

- overspecifies computationally-intensive systems,
- 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
- Costly to dynamically schedule

3.5. Finite State Machines

Finite state machines (FSMs) are describe the transitional behavior of sequential circuits. Finite state machines have two primary representations: state transition graphs and state transition tables. Both representations are very similar in that they show the output and the new state for every possible input and previous state. Specifying FSMs is important in formalizing the design of network protocols, embedded controllers, transceivers, and source coders. Traditional (flat) 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.

Traditional FSMs do not support concurrency or hierarchy. 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 [11]. 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 a lecture on introducing graph theory, we introduce the scheduling problem. 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 optimal scheduling problem is NP-complete— no algorithm can schedule a graph in polynomial time in the size of the graph. In the Synchronous/Reactive model, however, optimal scheduling in the sense of minimum number of block evaluations can always be performed quadratic time. In practice, electronic

design automation tools use polynomial-time heuristics to find good but suboptimal schedules.

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 EDA tools: Ptolemy from UC Berkeley and Advanced Design System (ADS) from Agilent 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.

Agilent EEsof ADS is a integrated, end-to-end signal path design solution for communications products. Agilent EEsof ADS products include digital signal processing design and synthesis, RF and high-frequency circuit, electromagnetic, and system simulators, schematic capture, layout tools, libraries, and device modeling systems. Students use this tool to design, simulate, and synthesis end-to-end systems. Among the circuit simulation modules, system simulation modules, and other simulation modules, the course focuses on the system simulation modules.

Agilent EEsof ADS has two system simulation modules:

1. HP Ptolemy
2. RF Systems

The HP Ptolemy Simulator is a system level design tool based on a hybrid of dataflow and timed-synchronous dataflow models. This tool allows simulation and design of combined DSP, RF, and analog systems. For example, it is possible to simulate RF mixers, analog filters, and amplifiers of a

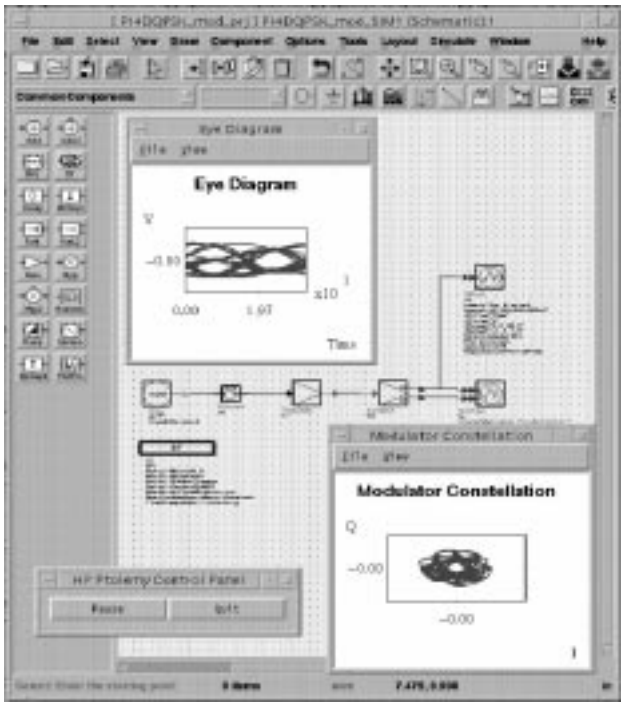

Figure 2: A snapshot of a $\pi/4$ DQPSK modulator simulation in Agilent EEsof Advanced Design System.

communication system in the same environment with the digital signal processing of the baseband signal.

In order to support the cosimulation of RF and DSP modules, HP Ptolemy uses an extensive set of models. The models are divided into three groups:

1. numeric models simulate DSP and digital algorithms and models for baseband designs.
2. timed models are for analog and RF models including RF effects.
3. synthesizable models are used in code generation.

In addition to the models a simulation engine and signal representations are major building blocks of a simulation environment. In the case of a cosimulation environment a collection of these components are required. For example, the Agilent EEsof ADS circuit simulator uses models of computation such as linear, harmonic balance, circuit envelope, and convolution. In order to handle transient behavior a high frequency SPICE simulator is used. For DSP algorithms dataflow models are effective. Dataflow models such as SDF, BDF, and DF are ideal for high level abstraction of a possibly complex DSP algorithm. The simulation engine fires each block of computation based on the model of computation and the schedule generated by the scheduler statically or dynamically.

The cosimulation is implemented in a hierarchical manner. Circuits are designed on the circuit schematic which are simulated and tested using RF/analog simulation engines. After adding appropriate interfacing port to the verified circuit it can be used as a building block in the DSP schematic. Every time the DSP simulator fires this building block the circuit simulator runs the RF/analog simulator

and returns the results based on the current inputs to the DSP simulator.

Every time data is passed from one domain to the other the signal being exchanged needs to be modified according to the new domain's data type. Both the DSP simulator and the RF/analog simulator have a notion of time. Time is discrete in the DSP simulators, whereas time has various levels of resolution in the RF/analog simulator. Timed-synchronous dataflow model are required in order to keep the time stamp information in the DSP level.

6. TRANSITION TO THE SENIOR LEVEL

The course is currently geared for a first-year graduate student who has taken courses in signals and systems, embedded processor programming, data structures and algorithms, and object-oriented programming. At UT Austin, UC Berkeley, Georgia Tech, and possibly many other universities, a senior electrical engineering student could have taken these pre-requisite courses. Nonetheless, as an undergraduate course, less time should be spent on the mathematical framework underlying the models of computation and more time should be devoted to a wider selection of topics. Hence, the textbook on *Languages for Embedded Systems* would be appropriate, but the other two previously used textbooks [12, 13] would not be appropriate.

In the course, students analyze material by completing homework assignments and studying for tests. They synthesize material by completing a literature survey, a computer implementation, and a final report. The literature survey and final report are presented orally. The final grade is based on two midterms (20% each), homework (10%), literature survey (25%), and final report (25%). There is no final exam. This format could work for a senior elective. The 50% project would still be the catalyst for the students to gain extra hands-on experience with system-level EDA tools, but the scope of the implementation should be reduced to an appropriate level.

7. CONCLUSION

In some courses, students learn how to model, design, and optimize specific systems such as signal processing, communication and control. Since today's embedded systems may combine all of these subsystems, a system-level design approach is needed in order to control system-level constraints and achieve global optimization. This course surveys modern methods for specifying algorithms, simulating systems, and mapping specifications onto embedded systems. The specification is decoupled from the implementation so that alternative implementations can be considered for the same specification. Specification is by a hierarchical combination of implementation-unbiased models of computation. The models of computation are formal in that they have a solid mathematical basis; they support heterogeneity in that they can be composed to characterize heterogeneous systems; and they are general in that both hardware and software can be synthesized from them. The course introduces the technologies used in the design and implementation of programmable embedded systems, including electronic devices such as programmable processors, cores, memories, and dedicated and configurable hardware, and software tools such as compilers, schedulers, code generators, and system-level design tools.

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 Agilent EEsof to evaluate models of computation and scheduling algorithms. Many also use these environments to implement their projects.

8. ACKNOWLEDGMENTS

The Embedded Software Systems course draws material from the Fall 1996 course at UC Berkeley by Prof. Edward A. Lee entitled *Specification and Modeling of Reactive Real-Time Systems*:

http://ptolemy.eecs.berkeley.edu/~eal/ee290n/

REFERENCES

[1] H. J. Reekie, *Realtime DSP: The TMS32030 Course*. ptolemy.eecs.berkeley.edu/~johnr/tutorials/c30.html, 1994.

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

[3] S. M. Kuo and G. D. Miller, "An innovative course emphasizing real-time digital signal processing applications," *IEEE Trans. on Education*, vol. 39, pp. 109–113, May 1996.

[4] *Magic*, http://research.compaq.com/wrl/projects/magic.

[5] *Synopsys*, http://www.synopsys.com/.

[6] *Mentor Graphics*, http://www.mentorgraphics.com/.

[7] *Cadence*, http://www.cadence.com/.

[8] B. L. Evans and G. Arslan, "Raising the level of abstraction: A signal processing system design course," in *Proc. IEEE-EURASIP Int. Work. on Nonlinear Signal and Image Proc.*, vol. 2, pp. 569–573, June 1999.

[9] *Ptolemy Classic*, http://ptolemy.eecs.berkeley.edu.

[10] *Agilent Advanced Design System*, http://contact.tm.agilent.com/tmo/hpeesof/products/ads/.

[11] S. A. Edwards, *Languages for Digital Embedded Systems*. Kluwer Academic Press, 2000.

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

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

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