Embedded Software

EDWARD A. LEE

Department of Electrical Engineering and Computer Science
University of California—Berkeley
518 Cory Hall
Berkeley, CA 94720-1770
USA
eal@eecs.berkeley.edu

Abstract

The science of computation has systematically abstracted away the physical world. Embedded software systems, however, engage the physical world. Time, concurrency, liveness, robustness, continuums, reactivity, and resource management must be remarried to computation. Prevailing abstractions of computational systems leave out these "nonfunctional" aspects. This chapter explains why embedded software is not just software on small computers, and why it therefore needs fundamentally new views of computation. It suggests component architectures based on a principle called "actor-oriented design," where actors interact according to a model of computation, and describes some models of computation that are suitable for embedded software. It then suggests that actors can define interfaces that declare dynamic aspects that are essential to embedded software, such as temporal properties. These interfaces can be structured in a "system-level-type system" that supports the sort of design-time- and run-time-type checking that conventional software benefits from.

1. What is Embedded Software?

Deep in the intellectual roots of computation is the notion that software is the realization of mathematical functions as procedures. These functions map a body of input data into a body of output data. The mechanism used to carry out the procedure is not nearly as important as the abstract properties of the function. In fact, we can reduce the mechanism to seven operations on a machine (the famous Turing machine) with an infinite tape capable of storing zeros and ones [1]. This mechanism is, in theory, as good as any other mechanism, and therefore, the significance of the software is not affected by the mechanism.

Embedded software is not like that. Its principal role is not the transformation of data, but rather the interaction with the physical world. It executes on machines that are not, first and foremost, computers. They are cars, airplanes, telephones, audio equipment, robots, appliances, toys, security systems, pacemakers, heart monitors, weapons, television sets, printers, scanners, climate control systems, manufacturing systems, and so on.

Software with a principal role of interacting with the physical world must, of necessity, acquire some properties of the physical world. It takes time. It consumes power. It does not terminate (unless it fails). It is not the idealized procedures of Alan Turing.

Computer science has tended to view this physicality of embedded software as messy. Consequently, the design of embedded software has not benefited from the richly developed abstractions of the 20th century. Instead of using object modeling, polymorphic-type systems, and automated memory management, engineers write assembly code for idiosyncratic digital signal processors (DSPs) that can do finite impulse response filtering in one (deterministic) instruction cycle per tap.

The engineers that write embedded software are rarely computer scientists. They are experts in the application domain with a good understanding of the target architectures they work with. This is probably appropriate. The principal role of embedded software is interaction with the physical world. Consequently, the designer of that software should be the person who best understands that physical world. The challenge to computer scientists, should they choose to accept it, is to invent better abstractions for that domain expert to do her job.

Today's domain experts may resist such help. In fact, their skepticism is well warranted. They see Java programs stalling for one-third of a second to perform garbage collection and update the user interface, and they envision airplanes falling out of the sky. The fact is that the best-of-class methods offered by computer scientists today are, for the most part, a poor match to the requirements of embedded systems.

At the same time, however, these domain experts face a serious challenge. The complexity of their applications (and consequent size of their programs) is growing rapidly. Their devices now often sit on a network, wireless or wired. Even some programmable DSPs now run a TCP/IP protocol stack, and the applications are getting much more dynamic, with downloadable customization and migrating code. Meanwhile, reliability standards for embedded software remain very high, unlike general-purpose software. At a minimum, the methods used for general-purpose software require considerable adaptation for embedded software. At a maximum, entirely new abstractions that embrace physicality and deliver robustness are needed.

2. Just Software on Small Computers?

An arrogant view of embedded software is that it is just software on small computers. This view is naïve. Timeliness, concurrency, liveness, reactivity, and heterogeneity need to be an integral part of the programming abstractions. They are essential to the correctness of a program. It is not sufficient to realize the right mapping from input data to output data.

2.1 Timeliness

Time has been systematically removed from theories of computation. "Pure" computation does not take time, and has nothing to do with time. It is hard to overemphasize how deeply rooted this is in our culture. So-called "real-time" operating systems often reduce the characterization of a component (a process) to a single number, its priority. Even most "temporal" logics talk about "eventually" and "always," where time is not a quantifier, but rather a qualifier [2]. Attempts to imbue object-oriented design with real-time are far from satisfactory [3].

Much of the problem is that computation *does* take time. Computer architecture has been tending toward making things harder for the designers of embedded systems. A large part of the (architectural) performance gain in modern processors comes from statistical speedups such as elaborate caching schemes, speculative instruction execution, dynamic dispatch, and branch prediction. These techniques compromise the reliability of embedded systems. In fact, most embedded processors such as programmable DSPs and microcontrollers do not use these techniques. I believe that these techniques have such a big impact on average case performance that they are indispensable. However, software practitioners will have to find abstractions that regain control of time, or the embedded system designers will continue to refuse to use these processors.

The issue is not just that execution takes time. Even with infinitely fast computers, embedded software would still have to deal with time because the physical processes, with which it interacts, evolve over time.

2.2 Concurrency

Embedded systems rarely interact with only a single physical process. They must simultaneously react to stimulus from a network and from a variety of sensors, and at the same time, retain timely control over actuators. This implies that embedded software is concurrent.

In general-purpose software practice, management of concurrency is primitive. Threads or processes, semaphores, and monitors [4] are the classic tools for managing concurrency, but I view them as comparable to assembly language in abstraction. They are very difficult to use reliably, except by operating system experts. Only trivial designs are completely comprehensible (to most engineers). Excessively conservative rules of thumb dominate (such as always grab locks in the same order [5]). Concurrency theory has much to offer that has not made its way into widespread practice, but it probably needs adaptation for the embedded system context. For instance, many theories reduce concurrency to "interleavings," which trivialize time by asserting that all computations are equivalent to sequences of discrete timeless operations.

Embedded systems engage the physical world, where multiple things happen at once. Reconciling the sequentiality of software and the concurrency of the real world is a key challenge in the design of embedded systems. Classical approaches to concurrency in software (threads, processes, semaphore synchronization, monitors for mutual exclusion, rendezvous, and remote procedure calls) provide a good foundation, but are insufficient by themselves. Complex compositions are simply too hard to understand.

An alternative view of concurrency that seems much better suited to embedded systems is implemented in synchronous/reactive languages [6] such as Esterel [7], which are used in safety-critical real-time applications. In Esterel, concurrency is compiled away. Although this approach leads to highly reliable programs, it is too static for some networked embedded systems. It requires that mutations be handled more as incremental compilation than as process scheduling, and incremental compilation for these languages proves to be challenging. We need an approach somewhere in between that of Esterel and that of today's real-time operating systems, with the safety and predictability of Esterel and the adaptability of a real-time operating system.

2.3 Liveness

In embedded systems, liveness is a critical issue. Programs must not terminate or block waiting for events that will never occur. In the Turing view of computation, all nonterminating programs fall into an equivalence class that is implicitly deemed to be a class of defective programs. In embedded computing, however, terminating programs are defective. The term "deadlock" pejoratively describes premature termination of such systems. It is to be avoided at all costs.

In the Turing paradigm, given a sufficiently rich abstraction for expressing procedures, it is undecidable whether those procedures halt. This undecidability has been inconvenient because we cannot identify programs that fail to halt. Now it should be viewed as inconvenient because we cannot identify programs that fail to keep running.

Moreover, correctness cannot be viewed as getting the right final answer. It must take into account the timeliness of a continuing stream of partial answers, as well as other "nonfunctional" properties. A key part of the prevailing computation paradigm is that software is defined by the function it computes. The premise is that the function models everything interesting about the software. Even for the portions of embedded software that terminate (and hence have an associated "computable function"), this model is a poor match. A key feature of embedded software is its interaction with physical processes, via sensors and actuators. Nonfunctional properties include timing, power consumption, fault recovery, security, and robustness.

2.4 Interfaces

Software engineering has experienced major improvements over the past decade or so through the widespread use of object-oriented design. Object-oriented design is a *component technology*, in the sense that a large complicated design is composed of pieces that expose *interfaces* that abstract their own complexity.

The use of interfaces in software is not new. It is arguable that the most widely applied component technology based on interfaces is procedures. Procedures are finite computations that take predefined arguments and produce final results. Procedure libraries are marketable component repositories, and have provided an effective abstraction for complex functionality. Object-oriented design aggregates procedures with the data that they operate on (and renames the procedures "methods").

Procedures, however, are a poor match for many embedded system problems. Consider, for example, a speech coder for a cellular telephone. It is artificial to define the speech coder in terms of finite computations. It can be done of course. However, a speech coder is more like a process than a procedure. It is a nonterminating computation that transforms an unbounded stream of input data into an unbounded stream of output data. Indeed, a commercial speech coder component for cellular telephony is likely to be defined as a process that expects to execute on a dedicated signal processor. There is no widely accepted mechanism for packaging the speech coder in any way that it can safely share computing resources with other computations.

Processes, and their cousin, threads, are widely used for concurrent software design. Processes can be viewed as a component technology, where a multitasking operating system or multithreaded execution engine provides the framework that coordinates the components. Process interaction mechanisms, such as monitors, semaphores, and remote procedure calls, are supported by the framework. In this context, a process can be viewed as a component that exposes at its interface an ordered sequence of external interactions.

However, as a component technology, processes and threads are extremely weak. A composition of two processes is not a process (it no longer exposes at its interface an ordered sequence of external interactions). Worse, a composition of two processes is not a component of any sort that we can easily characterize. It is for this reason that concurrent programs built from processes or threads are so hard to get right. It is very difficult to talk about the properties of the aggregate because we have no ontology for the aggregate. We don't know what it is. There is no (understandable) interface definition.

Object-oriented interface definitions work well because of the type systems that support them. Type systems are one of the great practical triumphs of contemporary software. They do more than any other formal method to ensure correctness of (practical) software. Object-oriented languages, with their user-defined abstract data types, and their relationships between these types (inheritance, polymorphism) have had a big impact in both reusability of software (witness the Java class libraries) and the quality of software. Combined with design patterns [8] and object modeling [9], type systems give us a vocabulary for talking about larger structure in software than lines of code and procedures.

However, object-oriented programming talks only about static structure. It is about the syntax of procedural programs, and says nothing about their concurrency or dynamics. For example, it is not part of the type signature of an object that the initialize() method must be called before the fire() method. Temporal properties of an object (method x() must be invoked every 10 ms) are also not part of the type signature. For embedded software to benefit from a component technology, that component technology will have to include dynamic properties in interface definitions.

2.5 Heterogeneity

Heterogeneity is an intrinsic part of computation in embedded systems. They mix computational styles and implementation technologies. First, such systems are often a mixture of hardware and software designs, so that the embedded software interacts with hardware that is specifically designed to interact with it. Some of this hardware has continuous-time dynamics, which is a particularly poor match to prevailing computational abstractions.

Embedded systems also mix heterogeneous event handling styles. They interact with events occurring irregularly in time (alarms, user commands, sensor triggers, etc.) and regularly in time (sampled sensor data and actuator control signals). These events have widely different tolerances for timeliness of reaction. Today, they are intermingled in real-time software in ad hoc ways; for example, they might be all abstracted as periodic events, and rate-monotonic principles [10] might be used to assign priorities.

Perhaps because of the scientific training of most engineers and computer scientists, the tendency is to seek a grand-unified theory, the common model that subsumes everything as a special case, and that can, in principle, explain it all. We find it anathema to combine multiple programming languages, despite the fact that this occurs in practice all the time. Proponents of any one language are sure, absolutely sure, that their language is fully general. There is no need for any other, and if only the rest of the world would understand its charms, they would switch to using it. This view will never work for embedded systems, since languages are bound to fit better or worse for any given problem.

2.6 Reactivity

Reactive systems are those that react continuously to their environment at the speed of the environment. Harel and Pnueli [11] and Berry [12] contrast them with *interactive systems*, which react with the environment at their own speed, and *transformational systems*, which simply take a body of input data and transform it into a body of output data. Reactive systems have real-time constraints, and are frequently safety-critical, to the point that failures could result in loss of human life. Unlike transformational systems, reactive systems typically do not terminate (unless they fail).

Robust distributed networked reactive systems must be capable of adapting to changing conditions. Service demands, computing resources, and sensors may appear and disappear. Quality of service demands may change as conditions change. The system is therefore continuously being redesigned while it operates, and all the while it must not fail.

A number of techniques for providing more robust support for reactive system design than what is provided by real-time operating systems have emerged. The synchronous languages, such as Esterel [7], Lustre [13], Signal [14], and Argos [15], are reactive, have been used for applications where validation is important, such as safety-critical control systems in aircraft and nuclear power plants. Lustre, for example, is used by Schneider Electric and Aerospatiale in France. Use of these languages is rapidly spreading in the automotive industry, and support for them is beginning to appear on commercial EDA (electronic design automation) software.

Reactive systems must typically react simultaneously to multiple sources of stimulus. Thus, they are concurrent. The synchronous languages manage concurrency in a very different way than that found in real-time operating systems. Their mechanism makes much heavier use of static (compile-time) analysis of concurrency to guarantee behavior. However, compile-time analysis of concurrency has a serious drawback: it compromises modularity and precludes adaptive software architectures.

3. Limitations of Prevailing Software Engineering Methods

Construction of complex embedded software would benefit from component technology. Ideally, these components are reusable, and embody valuable expertise in one or more aspects of the problem domain. The composition must be meaningful, and ideally, a composition of components yields a new component that can be used to form other compositions. To work, these components need to be abstractions of the complex, domain-specific software that they encapsulate. They must hide the details, and expose only the essential external interfaces, with well-defined semantics.

3.1 Procedures and Object Orientation

A primary abstraction mechanism of this sort in software is the procedure (or in object-oriented culture, a method). Procedures are terminating computations. They take arguments, perform a finite computation, and return results. The real world, however, does not start, execute, complete, and return.

Object orientation couples procedural abstraction with data to get data abstraction. Objects, however, are passive, requiring external invocation of their methods. So-called "active objects" are more like an afterthought, requiring still a model of computation to have any useful semantics. The real world is active, more like processes than objects, but with a clear and clean semantics that is firmly rooted in the physical world.

So while object-oriented design has proven extremely effective in building large software systems, it has little to offer to address the specific problems of the embedded system designer.

A sophisticated component technology for embedded software will talk more about processes than procedures, but we must find a way to make these processes compositional, and to control their real-time behavior in predictable and understandable ways. It will talk about concurrency and the models of computation used to regulate interaction between components. And it will talk about time.

3.2 Hardware Design

Hardware design, of course, is more constrained than software by the physical world. It is instructive to examine the abstractions that have worked for hardware, such as synchronous design. The synchronous abstraction is widely used in hardware to build large, complex, and modular designs, and has recently been applied to software [6], particularly for designing embedded software.

Hardware models are conventionally constructed using hardware description languages such as Verilog and VHDL; these languages realize a discrete-event model of computation that makes time a first-class concept, information shared by all components. Synchronous design is done through a stylized use of these languages. Discrete-event models are often used for modeling complex systems, particularly in the context of networking, but have not yet (to my knowledge) been deployed into embedded system design.

Conceptually, the distinction between hardware and software, from the perspective of computation, has only to do with the degree of concurrency and the role of time. An application with a large amount of concurrency and a heavy temporal content might as well be thought of using hardware abstractions, regardless of how it is implemented. An application that is sequential and has no temporal behavior might as well be thought of using software abstractions, regardless of how it is implemented. The key problem becomes one of identifying the appropriate abstractions for representing the design.

3.3 Real-Time Operating Systems

Most embedded systems, as well as many emerging applications of desktop computers, involve real-time computations. Some of these have hard deadlines, typically involving streaming data and signal processing. Examples include communication subsystems, sensor and actuator interfaces, audio and speech processing subsystems, and video subsystems. Many of these require not just real-time throughput, but also low latency.

In general-purpose computers, these tasks have been historically delegated to specialized hardware, such as SoundBlaster cards, video cards, and modems. In embedded systems, these tasks typically compete for resources. As embedded systems become networked, the situation gets much more complicated, because the combination of tasks competing for resources is not known at design time.

Many such embedded systems incorporate a real-time operating system, which offers specialized scheduling services tuned to real-time needs, in addition to standard operating system services such as I/O. The schedules might be based on priorities, using for example the principles of rate-monotonic scheduling [10,16], or on deadlines. There remains much work to be done to improve the match between the assumptions of the scheduling principle (such as periodicity, in the case of rate-monotonic scheduling) and the realities of embedded systems. Because the match is not always good today, many real-time embedded systems contain hand-built, specialized microkernels for task scheduling. Such microkernels, however, are rarely sufficiently flexible to accommodate networked applications, and as the complexity of embedded applications grows, they will be increasingly difficult to design. The issues are not simple. Unfortunately, current practice often involves fine tuning priorities until a particular implementation seems to work. The result is fragile systems that fail when anything changes.

A key problem in scheduling is that most techniques are not compositional. That is, even if assurances can be provided for an individual component, there are no systematic mechanisms for providing assurances to the aggregate of two components, except in trivial cases. A chronic problem with priority-based scheduling, known as priority inversion, is one manifestation of this problem.

Priority inversion occurs when processes interact, for example, by using a monitor to obtain exclusive access to a shared resource. Suppose that a low-priority process has access to the resource, and is preempted by a medium-priority process. Then a high-priority process preempts the medium-priority process and attempts to gain access to the resource. It is blocked by the low-priority process, but the low-priority process is blocked by the presence of an executable process with higher priority, the medium-priority process. By this mechanism, the high-priority process cannot execute until the medium-priority process completes and allows the low-priority process to relinquish the resource.

Although there are ways to prevent priority inversion (priority inheritance and priority ceiling protocols, for example), the problem is symptomatic of a deeper failure. In a priority-based scheduling scheme, processes interact both through the scheduler and through the mutual exclusion mechanism (monitors) supported by the framework. These two interaction mechanisms together, however, have no coherent compositional semantics. It seems like a fruitful research goal to seek a better mechanism.

3.4 Real-Time Object-Oriented Models

Real-time practice has recently been extended to distributed component software in the form of real-time CORBA and related models [17] and real-time object-oriented modeling (ROOM) [18]. CORBA is fundamentally a distributed object-oriented approach based on remote procedure calls. Built upon this foundation of remote procedure calls are various services, including an event service that provides a publish-and-subscribe semantics. Real-time CORBA extends this further by associating priorities with event handling, and leveraging real-time scheduling for processing events in a timely manner. Real-time CORBA, however, is still based on prevailing software abstractions. Thus, for effective real-time performance, a programmer must specify various numbers, such as worst-case and typical execution times for procedures, cached and not. These numbers are hard to know precisely. Real-time scheduling is then driven by additional parameters such as periodicity, and then tweaked with semantically weak parameters called "importance" and "criticality." These parameters, taken together, amount to guesses, as their actual effect on system behavior is hard to predict except by experimentation.

4. Actor-Oriented Design

Object-oriented design emphasizes inheritance and procedural interfaces. We need an approach that, like object-oriented design, constructs complex applications by assembling components, but emphasizes concurrency and communication abstractions, and admits time as a first-class concept. I suggest the term *actor-oriented design* for a refactored software architecture, where instead of objects, the components are parameterized *actors* with *ports*. Ports and parameters define the interface of an actor. A port represents an interaction with other actors, but unlike a method, does not have call-return semantics. Its precise semantics depends on the model of computation, but conceptually it represents signaling between components.

There are many examples of actor-oriented frameworks, including Simulink (from The MathWorks), LabVIEW (from National Instruments), Easy 5x (from Boeing), SPW (the Signal Processing Worksystem, from Cadence), and Cocentric System studio (from Synopsys). The approach has not been entirely ignored by the software engineering community, as evidenced by ROOM [18] and some architecture description languages (ADLs, such as Wright [19]). Hardware design languages, such as VHDL, Verilog, and SystemC, are all actor-oriented. In the academic community, active objects and actors [20,21], timed I/O automata [22], Polis and Metropolis [23], Giotto [24], and Ptolemy and Ptolemy II [25] all emphasize actor orientation.

Agha uses the term "actors," which he defines to extend the concept of objects to concurrent computation [26a]. Agha's actors encapsulate a thread of control and have interfaces for interacting with other actors. The protocols used for this interface are called interaction patterns, and are part of the model of computation. My use of the term "actors" is broader, in that I do not require the actors to encapsulate a thread of control, but I share with Agha the notion of interaction patterns, which I call the "model of computation."

Agha argues that no model of concurrency can or should allow all communication abstractions to be directly expressed. He describes message passing as akin to "gotos" in their lack of structure. Instead, actors should be composed using an interaction policy. These more specialized interaction policies will form models of computation.

4.1 Abstract Syntax

It is useful to separate syntactic issues from semantic issues. An *abstract syntax* defines how a design can be decomposed into interconnected components, without being concerned with how a design is represented on paper or in a computer file (that is the concern of the *concrete syntax*). An abstract syntax is also not concerned with the meaning of the interconnections of components, nor even what a component is. A design is a set of components and relationships among them, where the relationships conform to this abstract syntax. Here, we describe the abstract syntax using informal diagrams that illustrate these sets and relations by giving use cases, although formalizing the abstract syntax is necessary for precision.

Consider the diagram in Fig. 1. This shows three components (actors), each with one port, and an interconnection between these ports mediated by a *relation*. This illustrates a basic abstract syntax. The abstract syntax says nothing about the meaning of the interconnection, but rather just merely that it exists. To be useful, the abstract syntax is typically augmented with hierarchy, where an actor is itself an aggregate of actors. It can be further elaborated with such features as ports supporting multiple links and relations representing multiple connections. An elaborate abstract syntax of this type is described in [25].

4.2 Concrete Syntaxes

The abstract syntax may be associated with any number of concrete syntaxes. For instance, an XML schema might be used to provide a textual representation of a structure [26b]. A visual editor may provide a diagrammatic syntax, like that shown in Fig. 2.

Actor-oriented design does not require visual syntaxes. However, visual depictions of systems have always held a strong human appeal, making them extremely effective in conveying information about a design. Many of the methods described in this chapter can use such depictions to completely and formally specify models. Visual syntaxes can be every bit as precise and complete as textual syntaxes, particularly when they are judiciously combined with textual syntaxes.

Visual representations of models have a mixed history. In circuit design, schematic diagrams used to be routinely used to capture all of the essential information needed to implement some systems. Today, schematics are usually replaced by text in hardware description languages such as VHDL or Verilog. In other contexts, visual representations have largely failed, for example, flowcharts for

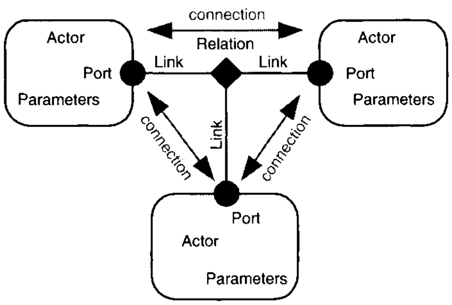
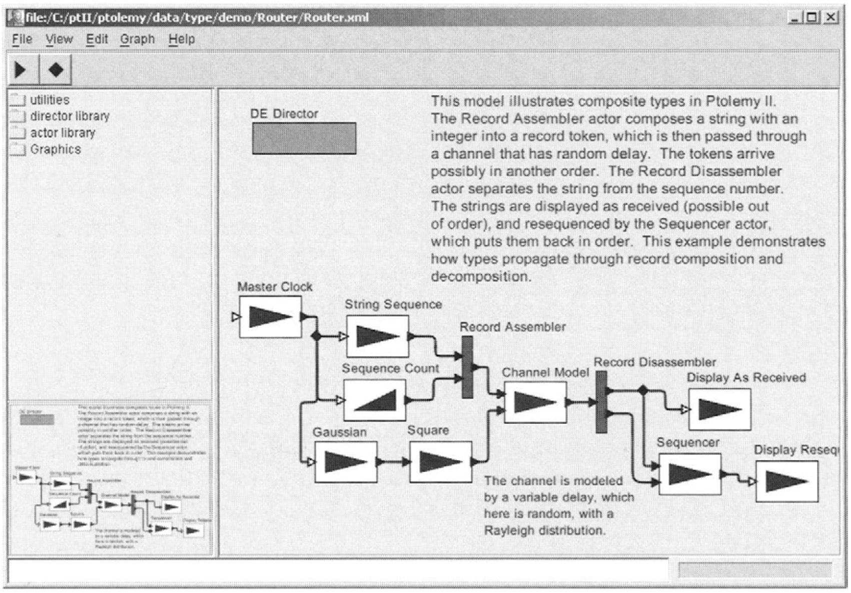


FIG. 1. Abstract syntax of actor-oriented designs.



This model illustrates composite types in Ptolemy II. The Record Assembler actor composes a string with an integer into a record token, which is then passed through a channel that has random delay. The tokens arrive possibly in another order. The Record Disassembler actor separates the string from the sequence number. The strings are displayed as received (possible out of order), and resequenced by the Sequencer actor, which puts them back in order. This example demonstrates how types propagate through record composition and decomposition.

The channel is modeled by a variable delay, which here is random, with a Rayleigh distribution.

FIG. 2. An example of a visual concrete syntax. This is the visual editor for Ptolemy II [25] called Vergil, designed by Steve Neuendorffer.

capturing the behavior of software. Recently, a number of innovative visual formalisms, including visual dataflow, hierarchical concurrent finite state machines, and object models, have been garnering support. The UML visual language for object modeling, for example, has been receiving a great deal of practical use [3,27].

4.3 Semantics

A semantics gives meaning to components and their interconnection. It states, for example, that a component is a process, and a connection represents communication between processes. Alternatively, a component may be a state, and a connection may represent a transition between states. In the former case, the semantics may restrict how the communication may occur. These semantic models can be viewed as architectural patterns [28], although for the purposes of this chapter, I will call them *models of computation*. One of my objectives here is to codify a few of the known models of computation that are useful for embedded software design.

Consider a family of models of computation where components are producers or consumers of data (or both). In this case, the ports acquire the property of being inputs, outputs, or both. Consider for example the diagram in Fig. 3.

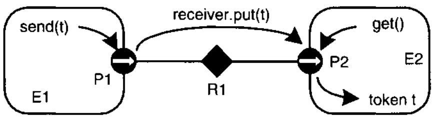


FIG. 3. Producer–consumer communication mechanism.

This diagram has two actors, one producer and one consumer. The diagram suggests a port that is an output by showing an outgoing arrow, and an input by showing an ingoing arrow. It also shows a simplified version of the Ptolemy II data transport mechanism [25]. The producer sends a token *t* (which encapsulates user data) via its port by calling a send() method on that port. This results in a call to the put() method of the receiver in the destination port. The destination actor retrieves the token by calling get() on the port. This mechanism, however, is polymorphic, in the sense that it does not specify what it means to call put() or get(). This depends on the model of computation.

A model of computation may be very broad or very specific. The more constraints there are, the more specific it is. Ideally, this specificity comes with benefits. For example, Unix pipes do not support feedback structures, and therefore cannot deadlock. Common practice in concurrent programming is that the components are threads that share memory and exchange objects using semaphores and monitors. This is a very broad model of computation with few benefits. In particular, it is hard to talk about the properties of an aggregate of components because an aggregate of components is not a component in the framework. Moreover, it is difficult to analyze a design in such a model of computation for deadlock or temporal behavior.

A model of computation is often deeply ingrained in the human culture of the designers that use it. It fades out of the domain of discourse. It can be argued that the Turing sequentiality of computation is so deeply ingrained in contemporary computer science culture that we no longer realize just how thoroughly we have banished time from computation. In a more domain-specific context, users of modeling languages such as Simulink rarely question the suitability of the semantics to their problem at hand. To such users, it does not "have semantics," it just "is." The key challenge in embedded software research is to invent or identify models of computation with properties that match the application domain well. One of the requirements is that time be central to the model.

4.4 Models of Computation

A model of computation can be thought of as the "laws of physics" that govern component interactions. It is the programmer's model, or the conceptual framework within which larger designs are constructed by composing components.

Design of embedded software will require models of computation that support concurrency. In practice, concurrency seriously complicates system design. No universal model of computation has yet emerged for concurrent computation (although some proponents of one approach or another will dispute this). By contrast, for sequential computation, Von Neumann provided a wildly successful universal abstraction. In this abstraction, a program consists of a sequence of transformations of the system state. In distributed systems, it is difficult to maintain a global notion of "system state," an essential part of the Von Neumann model, since many small state transformations are occurring simultaneously, in arbitrary order.

In networked embedded systems, communication bandwidth and latencies will vary over several orders of magnitude, even within the same system design. A model of computation that is well suited to small latencies (e.g., the synchronous hypothesis used in digital circuit design, where computation and communication take "zero" time) is usually poorly suited to large latencies, and vice versa. Thus, practical designs will almost certainly have to combine techniques.

It is well understood that effective design of concurrent systems requires one or more levels of abstraction above the hardware support. A hardware system with a shared memory model and transparent cache consistency, for example, still requires at least one more level of abstraction in order to achieve determinate distributed computation. A hardware system based on high-speed packet-switched networks could introduce a shared-memory abstraction above this hardware support, or it could be used directly as the basis for a higher level of abstraction. Abstractions that can be used include the event-based model of Java Beans, semaphores based on Dijkstra's P/V systems [29], guarded communication [30], rendezvous, synchronous message passing, active messages [31], asynchronous message passing, streams (as in Kahn process networks [32]), dataflow (commonly used in signal and image processing), synchronous/reactive systems [6], Linda [33], and many others.

These abstractions partially or completely define a model of computation. Applications are built on a model of computation, whether the designer is aware of this or not. Each possibility has strengths and weaknesses. Some guarantee determinacy, some can execute in bounded memory, and some are provably free from deadlock. Different styles of concurrency are often dictated by the application, and the choice of model of computation can subtly affect the choice of algorithms. While dataflow is a good match for signal processing, for example, it is a poor match for transaction-based systems, control-intensive sequential decision making, and resource management.

It is fairly common to support models of computation with language extensions or entirely new languages. Occam, for example, supports synchronous message passing based on guarded communication [30]. Esterel [7], Lustre [13], Signal [14], and Argos [15] support the synchronous/reactive model. These languages, however, have serious drawbacks. Acceptance is slow, platforms are limited, support software is limited, and legacy code must be translated or entirely rewritten.

An alternative approach is to explicitly use models of computation for coordination of modular programs written in standard, more widely used languages. The system-level specification language SystemC for hardware systems, for example, uses this approach (see `http://systemc.org`). In other words, one can decouple the choice of programming language from the choice of model of computation. This also enables mixing such standard languages in order to maximally leverage their strengths. Thus, for example, an embedded application could be described as an interconnection of modules, where modules are written in some combination of C, Java, and VHDL. Use of these languages permits exploiting their strengths. For example, VHDL provides FPGA targeting for reconfigurable hardware implementations. Java, in theory, provides portability, migratability, and a certain measure of security. C provides efficient execution.

The interaction between modules could follow any of several principles, e.g., those of Kahn process networks [32]. This abstraction provides a robust interaction layer with loosely synchronized communication and support for mutable systems (in which subsystems come and go). It is not directly built into any of the underlying languages, but rather interacts with them as an application interface. The programmer uses them as a design pattern [8] rather than as a language feature. Larger applications may mix more than one model of computation. For example, the interaction of modules in a real-time, safety-critical subsystem might follow the synchronous/reactive model of computation, while the interaction of this subsystem with other subsystems follows a process networks model. Thus, domain-specific approaches can be combined.

5. Examples of Models of Computation

There are many models of computation, each dealing with concurrency and time in different ways. In this section, I outline some of the most useful models for embedded software. All of these will lend a semantics to the same abstract syntax shown in Fig. 1.

5.1 Dataflow

In dataflow models, actors are atomic (indivisible) computations that are triggered by the availability of input data. Connections between actors represent the flow of data from a producer actor to a consumer actor. Examples of commercial frameworks that use dataflow models are SPW (signal processing worksystem, from Cadence) and LabVIEW (from National Instruments).

Synchronous dataflow (SDF) is a particularly restricted special case with the extremely useful property that deadlock and boundedness are decidable [34–38]. Boolean dataflow (BDF) is a generalization that sometimes yields to deadlock and boundedness analysis, although fundamentally these questions remain undecidable [39]. Dynamic dataflow (DDF) uses only run-time analysis, and thus makes no attempt to statically answer questions about deadlock and boundedness [40–42].

A small but typical example of an embedded software application modeled using SDF is shown in Fig. 4. That example shows a sound synthesis algorithm that consists of four actors in a feedback loop. The algorithm synthesizes the sound of a plucked string instrument, such as a guitar, using the well-known Karplus–Strong algorithm.

5.2 Time Triggered

Some systems with timed events are driven by *clocks*, which are signals with events that are repeated indefinitely with a fixed period. A number of software frameworks and hardware architectures have evolved to support this highly regular style of computation.

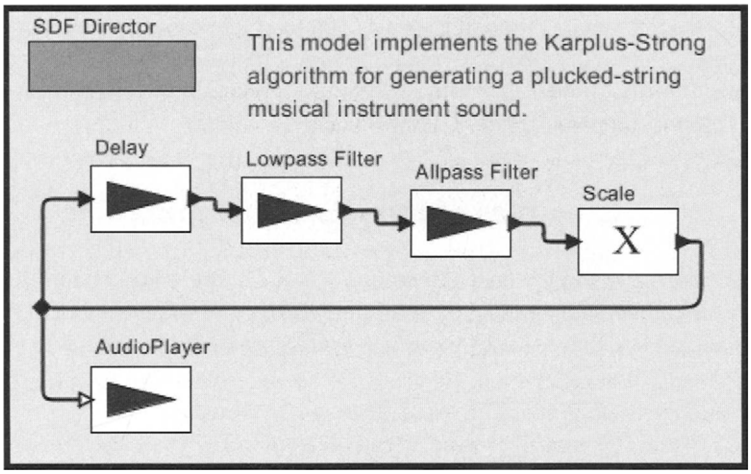

SDF Director
This model implements the Karplus-Strong algorithm for generating a plucked-string musical instrument sound.
Delay
Lowpass Filter
Allpass Filter
Scale
X
AudioPlayer

FIG. 4. A synchronous dataflow model implemented in the SDF domain (created by Stephen Neuendorffer) of Ptolemy II [25]. This model uses the audio library created by Brian Vogel.

The *time-triggered architecture* [43] is a hardware architecture supporting such models. The TTA takes advantage of this regularity by statically scheduling computations and communications among distributed components.

In hardware design, *cycle-driven simulators* stimulate computations regularly according to the clock ticks. This strategy matches synchronous hardware design well, and yields highly efficient simulations for certain kinds of designs. In the Scenic system [44], for example, components are processes that run indefinitely, stall to wait for clock ticks, or stall to wait for some condition on the inputs (which are synchronous with clock ticks). Scenic also includes a clever mechanism for modeling preemption, an important feature of many embedded systems. Scenic has evolved into the SystemC specification language for system-level hardware design (see `http://systemc.org`).

The Giotto programming language [24] provides a time-triggered software abstraction which, unlike the TTA or cycle-driven simulation, is hardware independent. It is intended for embedded software systems where periodic events dominate. It combines with finite-state machines (see below) to yield modal models that can be quite expressive. An example of a helicopter controller in Giotto is described in [45].

Discrete-time models of computation are closely related. These are commonly used for digital signal processing, where there is an elaborate theory that handles the composition of subsystems. This model of computation can be generalized to support multiple sample rates. In either case, a global clock defines the discrete points at which signals have values (at the ticks).

5.3 Synchronous/Reactive

In the synchronous/reactive (SR) model of computation [6], connections between components represent data values that are aligned with global clock ticks, as with time-triggered approaches. However, unlike time-triggered and discrete-time approaches, there is no assumption that all (or even most) signals have a value at each time tick. This model efficiently deals with concurrent models with irregular events. The components represent relations between input and output values at each tick, allowing for absences of value, and are usually partial functions with certain technical restrictions to ensure determinacy. Sophisticated compiler techniques yield extremely efficient execution that can reduce all concurrency to a sequential execution. Examples of languages that use the SR model of computation include Esterel [7], Signal [14], and Lustre [46].

An example of an application for which the synchronous reactive model is ideally suited is the management of a token-ring protocol for media access control, described in [9]. In this application, a token circulates in a round-robin fashion among users of a communication medium. When a user makes a request for access, if the user has the token, access is granted immediately. If not, then access may still be granted if the current holder of the token does not require access. The SR realization of this protocol yields predictable, deterministic management of access. This application benefits from the SR semantics because it includes instantaneous dialog and convergence to a fixed point (which determines who gets access when there is contention).

SR models are excellent for applications with concurrent and complex control logic. Because of the tight synchronization, safety-critical real-time applications are a good match. However, also because of the tight synchronization, some applications are overspecified in the SR model, which thus limits the implementation alternatives and makes distributed systems difficult to model. Moreover, in most realizations, modularity is compromised by the need to seek a global fixed point at each clock tick.

5.4 Discrete Events

In discrete-event (DE) models of computation, the connections represent sets of events placed on a time line. An event consists of a value and time stamp. This model of computation is popular for specifying hardware and for simulating telecommunications systems, and has been realized in a large number of simulation environments, simulation languages, and hardware description languages, including VHDL and Verilog. Like SR, there is a globally consistent notion of time, but unlike SR time has a metric, in that the time between events has significance.

DE models are often used in the design of communication networks. Figure 2 above gives a very simple DE model that is typical of this usage. That example constructs packets and routes them through a channel model. In this case, the channel model has the feature that it may reorder the packets. A sequencer is used to reconstruct the original packet order.

DE models are also excellent descriptions of concurrent hardware, although increasingly the globally consistent notion of time is problematic. In particular, it overspecifies (or overmodels) systems where maintaining such a globally consistent notion is difficult, including large VLSI chips with high clock rates, and networked distributed systems. A key weakness is that it is relatively expensive to implement in software, as evidenced by the relatively slow simulators.

5.5 Process Networks

A common way of handling concurrent software is where components are processes or threads that communicate by asynchronous, buffered message passing. The sender of the message need not wait for the receiver to be ready to receive the message. There are several variants of this technique, but I focus on one that ensures determinate computation, namely Kahn process networks [32].

In a Kahn process network (PN) model of computation, the connections represent sequences of data values (tokens), and the components represent functions that map input sequences into output sequences. Certain technical restrictions on these functions are necessary to ensure determinacy, meaning that the sequences are fully specified. Dataflow models are a special case of process networks that construct processes as sequences of atomic actor firings [47].

PN models are excellent for signal processing [48]. They are loosely coupled, and hence relatively easy to parallelize or distribute. They can be implemented efficiently in both software and hardware, and hence leave implementation options open. A key weakness of PN models is that they are awkward for specifying complicated control logic. Control logic is specified by routing data values.

5.6 Rendezvous

In synchronous message passing, the components are processes, and processes communicate in atomic, instantaneous actions called rendezvous. If two processes are to communicate, and one reaches the point first at which it is ready to communicate, then it stalls until the other process is ready to communicate. "Atomic" means that the two processes are simultaneously involved in the exchange, and that the exchange is initiated and completed in a single uninterruptable step. Examples of rendezvous models include Hoare's communicating sequential processes (CSP) [30] and Milner's calculus of communicating systems (CCS) [49]. This model of computation has been realized in a number of concurrent programming languages, including Lotos and Occam. Rendezvous models are particularly well matched to applications where resource sharing is a key element, such as client–server database models and multitasking or multiplexing of hardware resources. A key weakness of rendezvous-based models is that maintaining determinacy can be difficult. Proponents of the approach, of course, cite the ability to model nondeterminacy as a key strength.

Rendezvous models and PN both involve threads that communicate via message passing, synchronously in the former case and asynchronously in the latter. Neither model intrinsically includes a notion of time, which can make it difficult to interoperate with models that do include a notion of time. In fact, message events are partially ordered, rather than totally ordered as they would be were they placed on a time line.

Both models of computation can be augmented with a notion of time to promote interoperability and to directly model temporal properties (see, for example, [50]). In the Pamela system [51], threads assume that time does not advance while they are active, but can advance when they stall on inputs, outputs, or explicitly indicate that time can advance. By this vehicle, additional constraints are imposed on the order of events, and determinate interoperability with timed models of computation becomes possible. This mechanism has the potential of supporting low-latency feedback and configurable hardware.

5.7 Publish and Subscribe

In publish-and-subscribe models, connections between components are via named event streams. A component that is a consumer of such streams registers an interest in the stream. When a producer produces an event to such a stream, the consumer is notified that a new event is available. It then queries a server for the value of the event. Linda is a classic example of a fully elaborated publish-and-subscribe mechanism [52]. It has recently been reimplemented in JavaSpaces, from Sun Microsystems. An example of a distributed embedded software application using JavaSpaces is shown in Fig. 5.

5.8 Continuous Time

Physical systems can often be modeled using coupled differential equations. These have a natural representation in the abstract syntax of Fig. 1, where the connections represent continuous-time signals (functions of the time continuum). The components represent relations between these signals. The job of an execution environment is to find a fixed-point, i.e., a set of functions of time that satisfy all the relations.

Differential equations are excellent for modeling the physical systems with which embedded software interacts. Joint modeling of these physical systems and the software that interacts with them is essential to developing confidence in a design of embedded software. Such joint modeling is supported by such actor-oriented modeling frameworks as Simulink, Saber, VHDL-AMS, and Ptolemy II. A Ptolemy II continuous-time model is shown in Fig. 6.

5.9 Finite State Machines

All of the models of computation considered so far are concurrent. It is often useful to combine these concurrent models hierarchically with finite-state machines (FSMs) to get modal models. FSMs are different from any of the models we have considered so far in that they are strictly sequential. A component in this model is called a *state* or *mode*, and exactly one state is active at a time. The connections between states represent transitions, or transfer of control between states. Execution is a strictly ordered sequence of state transitions. Transition

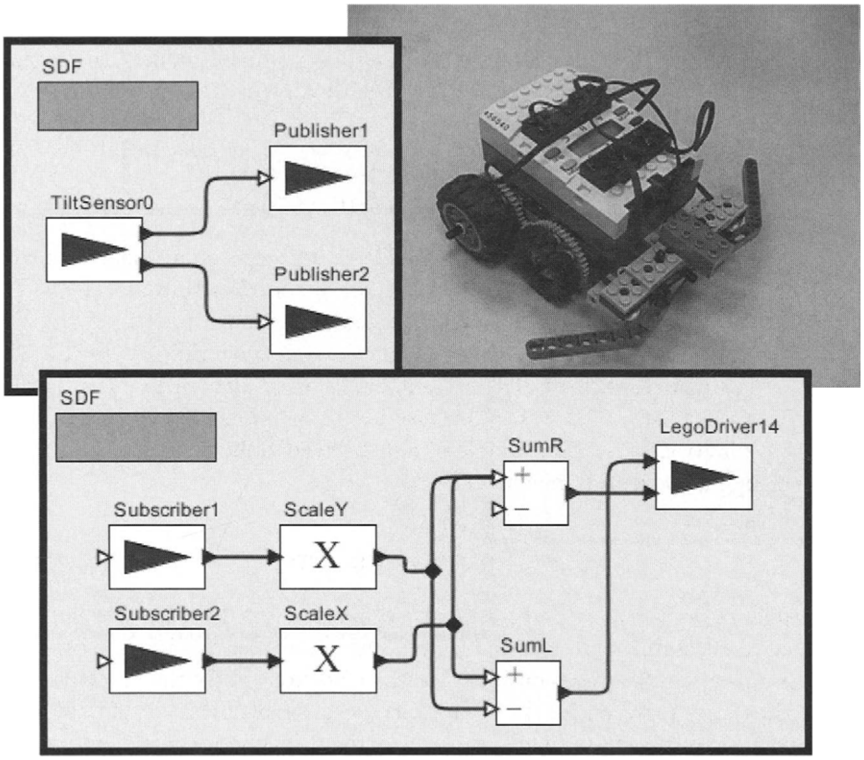


FIG. 5. A distributed embedded application using JavaSpaces combined with SDF to realize a publish-and-subscribe model of computation. The upper left model reads sensor data from a tilt sensor and publishes the data on the network. The lower model subscribes to the sensor data and uses it to drive the Lego robot at the upper right. This example was built by Jie Liu and Xiaojun Liu.

systems are a more general version, in that a given component may represent more than one system state (and there may be an infinite number of components).

FSM models are excellent for describing control logic in embedded systems, particularly safety-critical systems. FSM models are amenable to in-depth formal analysis, using for example model checking, and thus can be used to avoid surprising behavior. Moreover, FSMs are easily mapped to either hardware or software implementations.

FSM models have a number of key weaknesses. First, at a very fundamental level, they are not as expressive as the other models of computation described here. They are not sufficiently rich to describe all partial recursive functions. However, this weakness is acceptable in light of the formal analysis that becomes possible. Many questions about designs are decidable for FSMs and undecidable

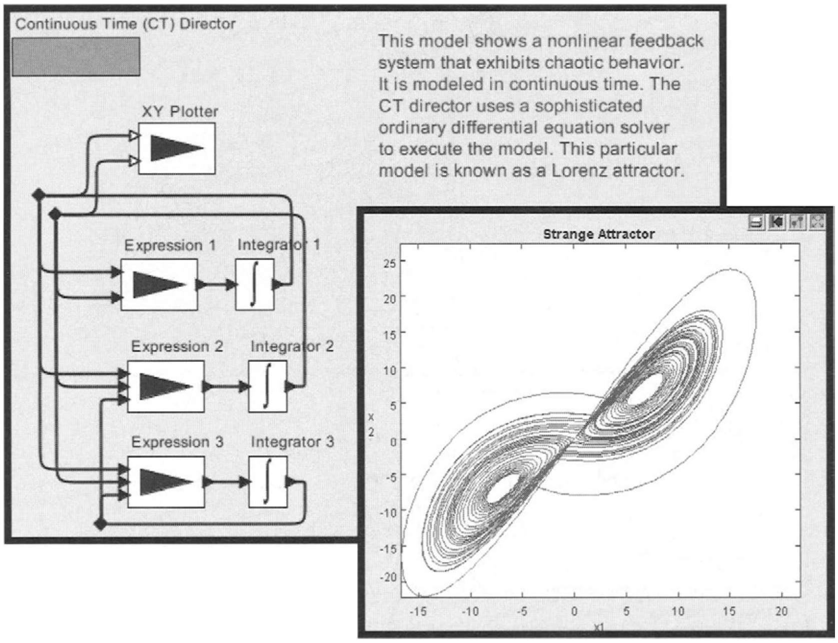

This model shows a nonlinear feedback system that exhibits chaotic behavior. It is modeled in continuous time. The CT director uses a sophisticated ordinary differential equation solver to execute the model. This particular model is known as a Lorenz attractor.

FIG. 6. A nonlinear feedback system modeled in the continuous-time (CT) domain in Ptolemy II. This model exhibits the chaotic behavior plotted at the right. This model and the CT domain were created by Jie Liu.

for other models of computation. Another key weakness is that the number of states can get very large even in the face of only modest complexity. This makes the models unwieldy.

The latter problem can often be solved by using FSMs in combination with concurrent models of computation. This was first noted by Harel, who introduced the Statecharts formalism. Statecharts combine synchronous/reactive modeling with FSMs [53a]. Statecharts have been adopted by UML for modeling the dynamics of software [3,27]. FSMs have also been combined with differential equations, yielding the so-called *hybrid systems* model of computation [53b].

FSMs can be hierarchically combined with a huge variety of concurrent models of computation. We call the resulting formalism "*charts" (pronounced "starcharts") where the star represents a wildcard [54].

Consider the model shown in Fig. 7. In that figure, component B is hierarchically refined by another model consisting of three components, c, d, and e. These latter three components are states of a state machine, and the connections between them are state transitions. States c and e are shown refined to concurrent models

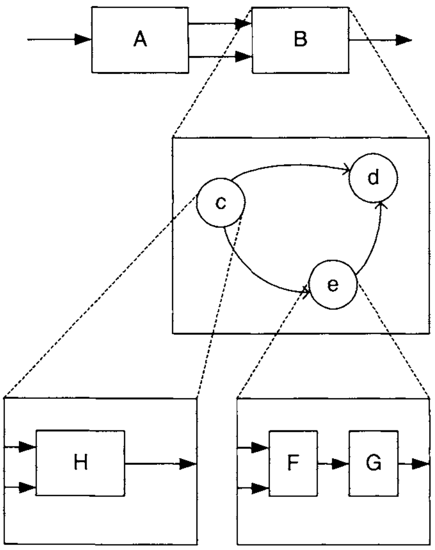


FIG. 7. Hierarchical composition of an FSM with concurrent models of computation.

themselves. The interpretation is that while the FSM is in state *c*, then component *B* is in fact defined by component *H*. While it is in state *e*, component *B* is defined by a composition of *F* and *G*.

In the figure, square boxes depict components in a concurrent model of computation, while circles depict states in a state machine. Despite the different concrete syntax, the abstract syntax is the same: components with interconnections. If the concurrent model of computation is SR, then the combination has Statechart semantics. If it is continuous time, then the combination has hybrid systems semantics. If it is PN, then the combination is similar to the SDL language [55]. If it is DE, then the combination is similar to Polis [23]. A hybrid system example implemented in Ptolemy II is shown in Fig. 8.

6. Choosing a Model of Computation

The rich variety of models of computation outlined above can be daunting to a designer faced with having to select them. Most designers today do not face this choice because they get exposed to only one or two. This is changing, however, as the level of abstraction and domain-specificity of design practice both rise. We expect that sophisticated and highly visual user interfaces will be needed to enable designers to cope with this heterogeneity.

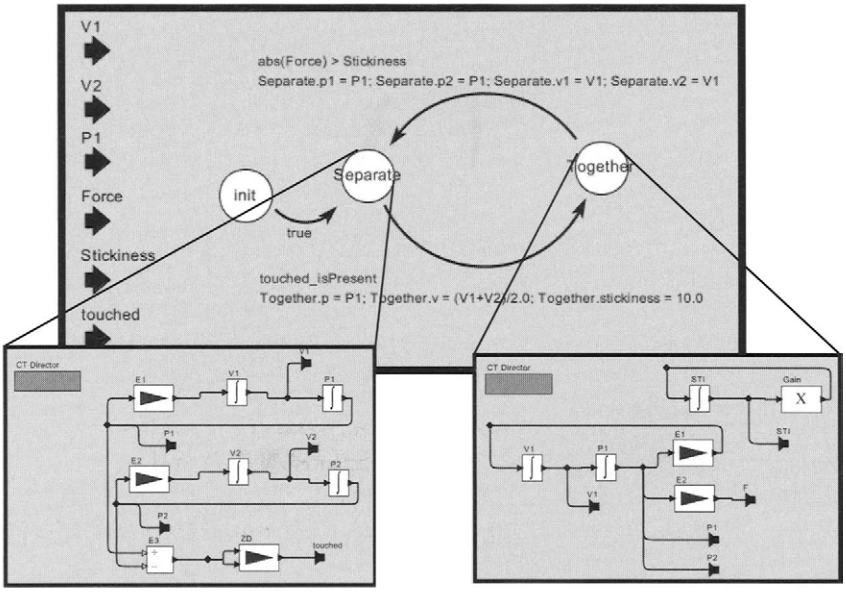

V1
V2
P1
Force
Stickiness
touched
abs(Force) > Stickiness
Separate.p1 = P1; Separate.p2 = P1; Separate.v1 = V1; Separate.v2 = V1
init
true
Separate
Together
touched_isPresent
Together.p = P1; Together.v = (V1+V2)/2.0; Together.stickiness = 10.0
CT Director
E1
V1
P1
V2
E2
P2
E3
ZD
touched
STI
Gain
X
F

FIG. 8. Hybrid system model in Ptolemy II, showing a hierarchical composition of a finite state machine (FSM) model and two continuous-time (CT) models. This example models a physical spring-mass system with two modes of operation. In the *Separate* mode, it has two masses on springs oscillating independently. In the *Together* mode, the two masses are struck together, and oscillate together with two springs. The model was created by Jie Liu and Xiaojun Liu.

An essential difference between concurrent models of computation is their modeling of time. Some are very explicit by taking time to be a real number that advances uniformly, and placing events on a time line or evolving continuous signals along the time line. Others are more abstract and take time to be discrete. Others are still more abstract and take time to be merely a constraint imposed by causality. This latter interpretation results in time that is partially ordered, and explains much of the expressiveness in process networks and rendezvous-based models of computation. Partially ordered time provides a mathematical framework for formally analyzing and comparing models of computation [56,57].

Many researchers have thought deeply about the role of time in computation. Benveniste and Le Guernic observe that in certain classes of systems, "the nature of time is by no means universal, but rather local to each subsystem, and consequently multiform" [14]. Lamport observes that a coordinated notion of time cannot be exactly maintained in distributed systems, and shows that a partial ordering is sufficient [58]. He gives a mechanism in which messages in an asynchronous system carry time stamps and processes manipulate these time stamps. We can then talk about processes having information or knowledge at a consistent cut, rather than "simultaneously." Fidge gives a related mechanism in which processes that can fork and join increment a counter on each event [59]. A partial ordering relationship between these lists of times is determined by process creation, destruction, and communication. If the number of processes is fixed ahead of time, then Mattern gives a more efficient implementation by using "vector time" [60]. All of this work offers ideas for modeling time.

How can we reconcile this multiplicity of views? A grand unified approach to modeling would seek a concurrent model of computation that serves all purposes. This could be accomplished by creating a *melange*, a mixture of all of the above. For example, one might permit each connection between components to use a distinct protocol, where some are timed and some not, and some are synchronous and some not, as done for example in ROOM [18] and SystemC 2.0 (`http://sys temc.org`). This offers rich expressiveness, but such a mixture may prove extremely complex and difficult to understand, and synthesis and validation tools would be difficult to design. In my opinion, such richly expressive formalisms are best used as foundations for more specialized models of computation. This, in fact, is the intent in SystemC 2.0 [61].

Another alternative would be to choose one concurrent model of computation, say the rendezvous model, and show that all the others are subsumed as special cases. This is relatively easy to do, in theory. Most of these models of computation are sufficiently expressive to be able to subsume most of the others. However, this fails to acknowledge the strengths and weaknesses of each model of computation. Process networks, for instance, are very good at describing the data dependencies in a signal processing system, but not as good at describing the associated control logic and resource management. Finite-state machines are good at modeling at least simple control logic, but inadequate for modeling data dependencies and numeric computation. Rendezvous-based models are good for resource management, but they overspecify data dependencies. Thus, to design interesting systems, designers need to use heterogeneous models.

Certain architecture description languages (ADLs), such as Wright [19] and Rapide [28], define a model of computation. The models are intended for describing the rich sorts of component interactions that commonly arise in software architecture. Indeed, such descriptions often yield good insights about design, but sometimes, the match is poor. Wright, for example, which is based on CSP, does not cleanly describe asynchronous message passing (it requires giving detailed descriptions of the mechanisms of message passing). I believe that what we really want are *architecture design languages* rather than architecture description languages. That is, their focus should not be on describing current practice, but rather on improving future practice. Wright, therefore, with its strong commitment to CSP, should not be concerned with whether it cleanly models asynchronous message passing. It should instead take the stand that asynchronous message passing is a bad idea for the designs it addresses.

7. Heterogeneous Models

Figure 4 shows a hierarchical heterogeneous combination of models of computation. A concurrent model at the top level has a component that is refined into a finite-state machine. The states in the state machine are further refined into a concurrent model of computation. Ideally, each concurrent model of computation can be designed in such a way that it composes transparently with FSMs, and, in fact, with other concurrent models of computation. In particular, when building a realization of a model of computation, it would be best if it did not need to be jointly designed with the realizations that it can compose with hierarchically.

This is a challenging problem. It is not always obvious what the meaning should be of some particular hierarchical combination. The semantics of various combinations of FSMs with various concurrency models are described in [54]. In Ptolemy II [25], the composition is accomplished via a notion called *domain polymorphism*.

The term "domain polymorphism" requires some explanation. First, the term "domain" is used in the Ptolemy project to refer to an implementation of a model of computation. This implementation can be thought of as a "language," except that it does not (necessarily) have the traditional textual syntax of conventional programming languages. Instead, it abides by a common abstract syntax that underlies all Ptolemy models. The term "domain" is a fanciful one, coming from the speculative notion in astrophysics that there are regions of the universe where the laws of physics differ. Such regions are called "domains." The model of computation is analogous to the laws of physics.

In Ptolemy II, components (called *actors*) in a concurrent model of computation implement an interface consisting of a suite of *action methods*. These methods define the execution of the component. A component that can be executed under the direction of any of a number of models of computation is called a domain polymorphic component. The component is not defined to operate with a particular model of computation, but instead has a well-defined behavior in several, and can be usefully used in several. It is *domain polymorphic,* meaning specifically that it has a well-defined behavior in more than one domain, and that the behavior is not necessarily the same in different domains. For example, the AddSubtract actor (shown as a square with a + and −) appears in Fig. 8, where it adds or subtracts continuous-time signals, and in Fig. 5, where it adds or subtracts streams.

In Ptolemy II, an application (which is called a "model") is constructed by composing actors (most of which are domain polymorphic), connecting them, and assigning a domain. The domain governs the interaction between components and the flow of control. It provides the execution semantics to the assembly of components. The key to hierarchically composing multiple models of computation is that an aggregation of components under the control of a domain should itself define a domain polymorphic component. Thus, the aggregate can be used as a component within a different model of computation. In Ptolemy II, this is how finite-state machine models are hierarchically composed with other models to get hybrid systems, Statechart-like models, and SDL-like models.

Domain polymorphic components in Ptolemy II simply need to implement a Java interface called Executable. This interface defines three phases of execution, an *initialization phase*, which is executed once, an *iteration phase*, which can be executed multiple times, and a *termination phase*, which is executed once. The iteration itself is divided into three phases also. The first phase, called *prefire*, can examine the status of the inputs and can abort the iteration or continue it. The prefire phase can also initiate some computation, if appropriate. The second phase, called *fire*, can also perform some computation, if appropriate, and can produce outputs. The third phase, called *postfire*, can commit any state changes for the component that might be appropriate.

To get hierarchical mixtures of domains, a domain must itself implement the Executable interface to execute an aggregate of components. Thus, it must define an initialization, iteration, and termination phase, and within the iteration phase, it must define the same three phases of execution.

The three-phase iteration has proven suitable for a huge variety of models of computation, including synchronous dataflow (SDF) [37], discrete events (DE) [62], discrete time (DT) [63], finite-state machines (FSM) [54], continuous-time (CT) [64], synchronous/reactive (SR), and Giotto (a time-triggered domain) [24]. All of these domains can be combined hierarchically.

Some domains in Ptolemy II have fixed-point semantics, meaning that in each iteration, the domain may repeatedly fire the components until a fixed point is found. Two such domains are continuous time (CT) [64] and synchronous/reactive (SR) [65,66]. The fact that a state update is committed only in the postfire phase of an iteration makes it easy to use domain-polymorphic components in such a domain.

Ptolemy II also has domains for which this pattern does not work quite as well. In particular, in the process networks (PN) domain [67] and communicating sequential processes (CSP) domain, each component executes in its own thread. These domains have no difficulty executing domain polymorphic components. They simply wrap in a thread a (potentially) infinite sequence of iterations.

However, aggregates in such domains are harder to encapsulate as domain polymorphic components, because it is hard to define an iteration for the aggregate. Since each component in the aggregate has its own thread of execution, it can be tricky to define the boundary points between iterations. This is an open issue that the Ptolemy project continues to address, and to which there are several candidate solutions that are applicable for particular problems.

8. Component Interfaces

The approach described in the previous section is fairly ad hoc. The Ptolemy project has constructed domains to implement various models of computation, most of which have entire research communities centered on them. It has then experimented with combinations of models of computation, and through trial and error, has identified a reasonable design for a domain polymorphic component interface definition. Can this ad hoc approach be made more systematic?

I believe that type system concepts can be extended to make this ad hoc approach more systematic. Type systems in modern programming languages, however, do not go far enough. Several researchers have proposed extending the type system to handle such issues as array bounds overruns, which are traditionally left to the run-time system [68]. However, many issues are still not dealt with. For example, the fact that *prefire* is executed before *fire* in a domain polymorphic component is not expressed in the type system.

At its root, a type system constrains what a component can say about its interface, and how compatibility is ensured when components are composed. Mathematically, type system methods depend on a partial order of types, typically defined by a subtyping relation (for user-defined types such as classes) or in more ad hoc ways (for primitive types such as double or int). They can be built from the robust mathematics of partial orders, leveraging, for example, fixed-point theorems to ensure convergence of type checking, type resolution, and type inference algorithms.

With this very broad interpretation of type systems, all we need is that the properties of an interface be given as elements of a partial order, preferably a complete partial order (CPO) or a lattice [18]. I suggest first that dynamic properties of an interface, such as the conventions in domain polymorphic component design, can be described using nondeterministic automata, and that the pertinent partial ordering relation is the simulation relation between automata. Preliminary work in this direction is reported in [69], which uses a particular automaton model called *interface automata* [29]. The result is called a *behavioral-type system*.

Behavioral-level types can be used without modifying the underlying languages, but rather by overlaying on standard languages design patterns that make these types explicit. Domain polymorphic components are simply those whose behavioral-level types are polymorphic.

Note that there is considerable precedent for such augmentations of the type system. For example, Lucassen and Gifford introduce state into functions using the type system to declare whether functions are free of side effects [70]. Martin-Löf introduces dependent types, in which types are indexed by terms [71]. Xi uses dependent types to augment the type system to include array sizes, and uses type resolution to annotate programs that do not need dynamic array bounds checking [68]. The technique uses singleton types instead of general terms [72] to help avoid undecidability. While much of the fundamental work has been developed using functional languages (especially ML [73]), there is no reason that I can see that it cannot be applied to more widely accepted languages.

8.1 On-line-type Systems

Static support for type systems gives the compiler responsibility for the robustness of software [74]. This is not adequate when the software architecture is dynamic. The software needs to take responsibility for its own robustness [75]. This means that algorithms that support the type system need to be adapted to be practically executable at run time.

ML is an early and well-known realization of a "modern type system" [1,76,77]. It was the first language to use type inference in an integrated way [78], where the types of variables are not declared, but are rather inferred from how they are used. The compile-time algorithms here are elegant, but it is not clear to me whether run-time adaptations are practical.

Many modern languages, including Java and C++, use declared types rather than type inference, but their extensive use of polymorphism still implies a need for fairly sophisticated type checking and type resolution. Type resolution allows for automatic (lossless) type conversions and for optimized run-time code, where the overhead of late binding can be avoided.

Type inference and type checking can be reformulated as the problem of finding the fixed point of a monotonic function on a lattice, an approach due to Dana Scott [79]. The lattice describes a partial order of types, where the ordering relationship is the subtype relation. For example, Double is a subtype of Number in Java. A typical implementation reformulates the fixed point problem as the solution of a system of equations [49] or of inequalities [80]. Reasonably efficient algorithms have been identified for solving such systems of inequalities [81], although these algorithms are still primarily viewed as part of a compiler, and not part of a run-time system.

Iteration to a fixed point, at first glance, seems too costly for on-line real-time computation. However, there are several languages based on such iteration that are used primarily in a real-time context. Esterel is one of these [7]. Esterel compilers synthesize run-time algorithms that converge to a fixed point at each clock of a synchronous system [14]. Such synthesis requires detailed static information about the structure of the application, but methods have been demonstrated that use less static information [65]. Although these techniques have not been proposed primarily in the context of a type system, I believe they can be adapted.

8.2 Reflecting Program Dynamics

Object-oriented programming promises software modularization, but has not completely delivered. The type system captures only static, structural aspects of software. It says little about the state trajectory of a program (its dynamics) and about its concurrency. Nonetheless, it has proved extremely useful, and through the use of reflection, is able to support distributed systems and mobile code.

Reflection, as applied in software, can be viewed as having an on-line model of the software within the software itself. In Java, for example, this is applied in a simple way. The static structure of objects is visible through the Class class and the classes in the reflection package, which includes Method, Constructor, and various others. These classes allow Java code to dynamically query objects for their methods, determine on-the-fly the arguments of the methods, and construct calls to those methods. Reflection is an integral part of Java Beans, mobile code, and CORBA support. It provides a run-time environment with the facilities for stitching together components with relatively intolerant interfaces.

However, static structure is not enough. The interfaces between components involve more than method templates, including such properties as communication protocols. To get adaptive software in the context of real-time applications, it will also be important to reflect the program state. Thus, we need reflection on the program dynamics.

In embedded software, this could be used, for example, to systematically realize fault detection, isolation, and recovery (FDIR). That is, if the declared dynamic properties of a component are violated at run time, the run-time-type checking can detect it. For example, suppose a component declares as part of its interface definition that it must execute at least once every 10 ms. Then a run-time-type checker will detect a violation of this requirement.

The first question becomes at what granularity to do this. Reflection intrinsically refers to a particular abstracted representation of a program. For example, in the case of static structure, Java's reflection package does not include finer granularity than methods.

Process-level reflection could include two critical facets, communication protocols and process state. The former would capture in a type system such properties as whether the process uses rendezvous, streams, or events to communicate with other processes. By contrast, Java Beans defines this property universally to all applications using Java Beans. That is, the event model is the only interaction mechanism available. If a component needs rendezvous, it must implement that on top of events, and the type system provides no mechanism for the component to assert that it needs rendezvous. For this reason, Java Beans seem unlikely to be very useful in applications that need stronger synchronization between processes, and thus it is unlikely to be used much beyond user interface design.

Reflecting the process state could be done with an automaton that simulates the program. (We use the term "simulates" in the technical sense of automata theory.) That is, a component or its run-time environment can access the "state" of a process (much as an object accesses its own static structure in Java), but that state is not the detailed state of the process, but rather the state of a carefully chosen automaton that simulates the application. Designing that automaton is then similar (conceptually) to designing the static structure of an object-oriented program, but represents dynamics instead of static structure.

Just as we have object-oriented languages to help us develop object-oriented programs, we would need state-oriented languages to help us develop the reflection automaton. These could be based on Statecharts, but would be closer in spirit to UML's state diagrams in that it would not be intended to capture all aspects of behavior. This is analogous to the object model of a program, which does not capture all aspects of the program structure (associations between objects are only weakly described in UML's static structure diagrams). Analogous to object-oriented languages, which are primarily syntactic overlays on imperative languages, a state-oriented language would be a syntactic overlay on an object-oriented language. The syntax could be graphical, as is now becoming popular with object models (especially UML).

Well-chosen reflection automata would add value in a number of ways. First, an application may be asked, via the network, or based on sensor data, to make some change in its functionality. How can it tell whether that change is safe? The change may be safe when it is in certain states, and not safe in other states. It would query its reflection automaton, or the reflection automaton of some gatekeeper object, to determine how to react. This could be particularly important in real-time applications. Second, reflection automata could provide a basis for verification via such techniques as model checking.

This complements what object-oriented languages offer. Their object model indicates safety of a change with respect to data layout, but they provide no mechanism for determining safety based on the state of the program.

When a reflection automaton is combined with concurrency, we get something akin to Statechart's concurrent, hierarchical FSMs, but with a twist. In Statecharts, the concurrency model is fixed. Here, any concurrency model can be used. We call this generalization "*charts," pronounced "starcharts," where the star represents a wildcard suggesting the flexibility in concurrency models [54]. Some variations of Statecharts support concurrency using models that are different from those in the original Statecharts [15,82]. As with Statecharts, concurrent composition of reflection automata provides the benefit of compact representation of a product automaton that potentially has a very large number of states. In this sense, aggregates of components remain components where the reflection automaton of the aggregate is the product automaton of the components, but the product automaton never needs to be explicitly represented.

Ideally, reflection automata would also inherit cleanly. Interface theories are evolving that promise to explain exactly how to do this [29].

In addition to application components being reflective, it will probably be beneficial for components in the run-time environment to be reflective. The run-time environment is whatever portion of the system outlives all application components. It provides such services as process scheduling, storage management, and specialization of components for efficient execution. Because it outlives all application components, it provides a convenient place for reflecting aspects of the application that transcend a single component or an aggregate of closely related components.

9. Frameworks Supporting Models of Computation

In this context, a *framework* is a set of constraints on components and their interaction, and a set of benefits that derive from those constraints. This is broader than, but consistent with the definition of frameworks in object-oriented design [83]. By this definition, there are a huge number of frameworks, some of which are purely conceptual, cultural, or even philosophical, and some of which are embodied in software. Operating systems are frameworks where the components are programs or processes. Programming languages are frameworks where the components are language primitives and aggregates of these primitives, and the possible interactions are defined by the grammar. Distributed component middleware such as CORBA [17] and DCOM are frameworks. Synchronous digital hardware design principles are a framework. Java Beans form a framework that is particularly tuned to user interface construction. A particular class library and policies for its use is a framework [83].

For any particular application domain, some frameworks are better than others. Operating systems with no real-time facilities have limited utility in embedded systems, for example.

In order to obtain certain benefits, frameworks impose constraints. As a rule, stronger benefits come at the expense of stronger constraints. Thus, frameworks may become rather specialized as they seek these benefits.

The drawback with specialized frameworks is that they are unlikely to solve all the framework problems for any complex system. To avoid giving up the benefits of specialized frameworks, designers of these complex systems will have to mix frameworks heterogeneously. Of course, a framework within which to heterogeneously mix frameworks is needed. The design of such a framework is the purpose of the Ptolemy project [25]. Each domain, which implements a model of computation, offers the designer a specialized framework, but domains can be mixed hierarchically using the concept of domain polymorphism.

A few other research projects have also heterogeneously combined models of computation. The Gravity system and its visual editor Orbit, like Ptolemy, provide a framework for heterogeneous models [84]. A model in a domain is called a facet, and heterogeneous models are multifacetted designs [85]. Jourdan *et al.* have proposed a combination of Argos, a hierarchical finite-state machine language, with Lustre [13], which has a more dataflow flavor, albeit still within a synchronous/reactive concurrency framework [86]. Another interesting integration of diverse semantic models is done in Statemate [87], which combines activity charts with statecharts. This sort of integration has more recently become part of UML. The activity charts have some of the flavor of a process network.

10. Conclusions

Embedded software requires a view of computation that is significantly different from the prevailing abstractions in computation. Because such software engages the physical world, it has to embrace time and other nonfunctional properties. Suitable abstractions compose components according to a model of computation. Models of computation with stronger formal properties tend to be more specialized. This specialization limits their applicability, but this limitation can be ameliorated by hierarchically combining heterogeneous models of computation. System-level types capture key features of components and their interactions through a model of computation, and promise to provide robust and understandable composition technologies.

ACKNOWLEDGMENTS

This chapter distills the work of many people who have been involved in the Ptolemy Project at Berkeley. Most notably, the individuals who have directly contributed ideas are Shuvra S. Bhattacharyya, John Davis II, Johan Eker, Chamberlain Fong, Christopher Hylands, Joern Janneck, Jie Liu, Xiaojun Liu, Stephen Neuendorffer, John Reekie, Farhana Sheikh, Kees Vissers, Brian K. Vogel, Paul Whitaker, and Yuhong Xiong.

The Ptolemy Project is supported by the Defense Advanced Research Projects Agency (DARPA), the MARCO/DARPA Gigascale Silicon Research Center (GSRC), the State of

California MICRO program, and the following companies: Agilent Technologies, Cadence Design Systems, Hitachi, and Philips.

References

[1] Turing, A. M. (1936). "On computable numbers with an application to the Entscheidungsproblem." *Proceedings of the London Mathematical Society*, **42**, 230–265.

[2] Manna, Z., and Pnueli, A. (1991). *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, Berlin.

[3] Douglass, B. P. (1998). *Real-Time UML*. Addison-Wesley, Reading, MA.

[4] Dijkstra, E. (1968). "Cooperating sequential processes." *Programming Languages* (E F. Genuys, Ed.). Academic Press. New York.

[5] Lea, D. (1997). *Concurrent Programming in JavaTM: Design Principles and Patterns*. Addison-Wesley, Reading MA.

[6] Benveniste, A., and Berry, G. (1991). "The synchronous approach to reactive and real-time systems." *Proceedings of the IEEE*, **79**, 1270–1282.

[7] Berry, G., and Gonthier, G. (1992). "The Esterel synchronous programming language: Design, semantics, implementation." *Science of Computer Programming*, **19**, 87–152.

[8] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

[9] Edwards, S. A., and Lee, E. A. (2001). "The semantics and execution of a synchronous block-diagram language," Technical Memorandum UCB/ERL, University of California—Berkeley. Available at `http://ptolemy.eecs.berkeley.edu/publications`.

[10] Liu, C., and Layland, J. (1973). "Scheduling algorithms for multiprogramming in a hard-real-time environment." *Journal of the ACM*, **20**, 46–61.

[11] Harel, D., and Pnueli, A. (1985). "On the development of reactive systems." *Logic and Models for Verification and Specification of Concurrent Systems*. Springer-Verlag, Berlin.

[12] Berry, G. (1989). "Real time programming: Special purpose or general purpose languages." *Information Processing* (G. Ritter, Ed.), Vol. 89, pp. 11–17. Elsevier Science, Amsterdam.

[13] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. (1991). "The synchronous data flow programming language LUSTRE." *Proc. IEEE*, **79**, 1305–1319.

[14] Benveniste, A., and Le Guernic, P. (1990). "Hybrid dynamical systems theory and the SIGNAL language." *IEEE Transactions on Automatic Control*, **35**, 525–546.

[15] Maraninchi, F. (1991). "The Argos Language: Graphical representation of automata and description of reactive systems." *Proceedings IEEE Workshop on Visual Languages*, Kobe, Japan, Oct.

[16] Klein, M. H., Ralya, T., Pollak, B., Obenza, R., and Harbour, M. G. (1993). *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic, Norwell, MA.

[17] Ben-Natan, R. (1995). *CORBA: A Guide to Common Object Request Broker Architecture.* McGraw-Hill, New York.

[18] Selic, B., Gullekson, G., and Ward, P. (1994). *Real-Time Object-Oriented Modeling.* Wiley, New York.

[19] Allen, R., and Garlan, D. (1994). "Formalizing architectural connection." *Proceedings of the 16th International Conference on Software Engineering (ICSE 94)*, pp. 71–80. IEEE Computer Society Press, Los Alamitos, CA.

[20] Agha, G. A. (1990). "Concurrent object-oriented programming." *Communications of the ACM*, **33**, 125–141.

[21] Agha, G. A. (1986). *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA.

[22] Lynch, N. A. (1996). *Distributed Algorithms.* Morgan Kaufmann, San Mateo, CA.

[23] Chiodo, M., Giusto, P., Hsieh, H., Jurecska, A., Lavagno, L., and Sangiovanni-Vincentelli, A. (1994). "A formal methodology for hardware/software co-design of embedded systems." *IEEE Micro*, **14**, 26–36.

[24] Henzinger, T. A., Horowitz, B., and Kirsch, C. M. (2001). "Giotto: A time-triggered language for embedded programming." *Proceedings of EMSOFT 2001*, Tahoe City, CA, Lecture Notes on Computer Science, 2211, pp. 166–184. Springer-Verlag, Berlin.

[25] Davis, II, J., Hylands, C., Kienhuis, B., Lee, E. A., Liu, J., Liu, X., Muliadi, L., Neuendorffer, S., Tsay, J., Vogel, B., and Xiong, Y. (2001). "Heterogeneous concurrent modeling and design in Java," Technical Memorandum UCB/ERL M01/12. Department of Electrical Engineering and Computer Science, University of California—Berkeley. Available at `http://ptolemy.eecs.berkeley.edu/publications`.

[26a] Agha, G. A. (1997). "Abstracting interaction patterns: A programming paradigm for open distributed systems." *Formal Methods for Open Object-Based Distributed Systems*, IFIP Transactions (E. Najm and J.-B. Stefani, Eds.) Chapman and Hall, London.

[26b] Lee, E. A., and Neuendorffer, S. (2000). "MoML—A modeling markup language in XML, Version 0.4," Technical Memorandum UCB/ERL M00/12. University of California—Berkeley. Available at `http://ptolemy.eecs.berkeley.edu/publications`.

[27] Eriksson, H.-E., and Penker, M. (1998). *UML Toolkit.* Wiley, New York.

[28] Luckham, D. C., and Vera, J. (1995). "An event-based architecture definition language." *IEEE Transactions on Software Engineering*, **21**, 717–734.

[29] de Alfaro, L., and Henzinger, T. A. (2001). "Interface theories for component-based design." *Proceedings of EMSOFT 2001*, Tahoe City, CA, Lecture Notes on Computer Science 2211, pp. 148–165. Springer-Verlag, Berlin.

[30] Hoare, C. A. R. (1978). "Communicating sequential processes." *Communications of the ACM*, **21**, 666–677.

[31] von Eicken, T., Culler, D. E., Goldstein, S. C., and Schauser, K. E. (1992). "Active messages: A mechanism for integrated communications and computation." *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia. Also available as Technical Report TR UCB/CSD 92/675, Computer Science Division, University of California—Berkeley.

[32] Kahn, G. (1974). "The semantics of a simple language for parallel programming." *Proceedings of the IFIP Congress 74*. North-Holland, Amsterdam.

[33] Carriero, N., and Gelernter, D. (1989). "Linda in context." *Communications of the ACM*, **32**, 444–458.

[34] Bhattacharyya, S. S., Murthy, P. K., and Lee, E. A. (1996). *Software Synthesis from Dataflow Graphs*. Kluwer Academic, Norwell, MA.

[35] Karp, R. M., and Miller, R. E. (1966). "Properties of a model for parallel computations: Determinacy, termination, queueing." *SIAM Journal*, **14**, 1390–1411.

[36] Lauwereins, R., Wauters, P., Adé, M., and Peperstraete, J. A. (1994). "Geometric parallelism and cyclo-static dataflow in GRAPE-II." *Proceedings 5th International Workshop on Rapid System Prototyping*, Grenoble, France.

[37] Lee, E. A., and Messerschmitt, D. G. (1987). "Synchronous data flow." *Proceedings of the IEEE*, **75**, 1235–1245.

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

[39] Buck, J. T. (1993). "Scheduling dynamic dataflow graphs with bounded memory using the token flow model," Tech. Report UCB/ERL 93/69, Ph.D. Dissertation. Department of Electrical Engineering and Computer Science, University of California—Berkeley. Available at `http://ptolemy.eecs.berkeley.edu/publications`.

[40] Jagannathan, R. (1992). "Parallel execution of GLU programs." Presented at *2nd International Workshop on Dataflow Computing, Hamilton Island*, Queensland, Australia.

[41] Kaplan, D. J., *et al.* (1987). "Processing Graph Method Specification Version 1.0," unpublished memorandum. Naval Research Laboratory, Washington DC.

[42] Parks, T. M. (1995). "Bounded scheduling of process networks." Technical Report UCB/ERL-95-105, Ph.D. Dissertation. Department of Electrical Engineering and Computer Science. University of California—Berkeley. Available at `http://ptolemy.eecs.berkeley.edu/publications`.

[43] Kopetz, H., Holzmann, M., and Elmenreich, W. (2000). "A universal smart transducer interface: TTP/A." *3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'2000)*.

[44] Liao, S., Tjiang, S., and Gupta, R. (1997). "An efficient implementation of reactivity for modeling hardware in the scenic design environment." *Proceedings of the Design Automation Conference (DAC 97)*, Anaheim, CA.

[45] Koo, T. J., Liebman, J., Ma, C., and Sastry, S. S. (2001). "Hierarchical approach for design of multi-vehicle multi-modal embedded software." *Proceedings of EMSOFT*

2001, Tahoe City, CA, Lecture Notes on Computer Science 2211, pp. 344–360. Springer-Verlag, Berlin.

[46] Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. A. (1987). "LUSTRE: A declarative language for programming synchronous systems." *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*, Munich, Germany.

[47] Lee, E. A., and Parks, T. M. (1995). "Dataflow process networks." *Proceedings of the IEEE*, **83**, 773–801.

[48] Lieverse, P., Van Der Wolf, P., Deprettere, E., and Vissers, K. (2001). "A methodology for architecture exploration of heterogeneous signal processing systems." *Journal of VLSI Signal Processing*, **29**, 197–207.

[49] Milner, R. (1978). "A theory of type polymorphism in programming." *Journal of Computer and System Sciences*, **17**, 348–375.

[50] Reed, G. M., and Roscoe, A. W. (1988). "A timed model for communicating sequential processes." *Theoretical Computer Science* **58**, 249–261.

[51] van Gemund, A. J. C. (1993). "Performance prediction of parallel processing systems: The PAMELA methodology." *Proceedings 7th International Conference on Supercomputing*, Tokyo.

[52] Ahuja, S., Carreiro, N., and Gelernter, D. (1986). "Linda and friends." *Computer*, **19**, 26–34.

[53a] Harel, D. (1987). "Statecharts: A visual formalism for complex systems." *Science of Computer Programming*, **8**, 231–274.

[53b] Henzinger, T. A. (1996). "The theory of hybrid automata." *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pp. 278–292. IEEE Computer Society Press, Los Alamitos, CA. Invited tutorial.

[54] Girault, A., Lee, B., and Lee, E. A. (1999). "Hierarchical finite state machines with multiple concurrency models." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **18**, 742–760.

[55] Saracco, S., Smith, J. R. W., and Reed, R. (1989). *Telecommunications Systems Engineering Using SDL*, North-Holland–Elsevier, Amsterdam.

[56] Lee, E. A., and Sangiovanni-Vincentelli, A. (1998). "A framework for comparing models of computation." *IEEE Transaction on Computer-Aided Design*, **17**, 1217–1229.

[57] Trotter, W. T. (1992). *Combinatorics and Partially Ordered Sets*. Johns Hopkins Univ. Press, Baltimore, MD.

[58] Lamport, L. (1978). "Time, clocks, and the ordering of events in a distributed system." *Communications of the ACM*, **21**, 558–565.

[59] Fidge, C. J. (1991). "Logical time in distributed systems." *Computer*, **24**, 28–33.

[60] Mattern, F. (1989). "Virtual time and global states of distributed systems." *Parallel and Distributed Algorithms* (M. Cosnard and P. Quinton, Eds.), pp. 215–226. North-Holland, Amsterdam.

[61] Swan, S. (2001). "An introduction to system level modeling in SystemC 2.0," draft report. Cadence Design Systems.

[62] Lee, E. A. (1999). "Modeling concurrent real-time processes using discrete events." *Annals of Software Engineering*, Special Volume on Real-Time Software Engineering, **7**, 25–45.

[63] Fong, C. (2001). "Discrete-time dataflow models for visual simulation in Ptolemy II," Memorandum UCB/ERL M01/9. Electronics Research Laboratory, University of California—Berkeley. Available at `http://ptolemy.eecs.berkeley.edu/publications`.

[64] Liu, J. (1998). "Continuous time and mixed-signal simulation in Ptolemy II," UCB/ERL Memorandum M98/74. Department of Electrical Engineering and Computer Science, University of California—Berkeley. Available at `http://ptolemy.eecs.berkeley.edu/publications`.

[65] Edwards, S. A. (1997). "The specification and execution of heterogeneous synchronous reactive systems," Technical Report UCB/ERL M97/31, Ph.D. thesis. University of California—Berkeley. Available at `http://ptolemy.eecs.berkeley.edu/publications`.

[66] Whitaker, P. (2001). "The simulation of synchronous reactive systems in Ptolemy II," Master's Report, Memorandum UCB/ERL M01/20. Electronics Research Laboratory, University of California—Berkeley. Available at `http://ptolemy.eecs.berkeley.edu/publications`.

[67] Goel, M. (1998). "Process networks in Ptolemy II," UCB/ERL Memorandum M98/69, University of California—Berkeley. Available at `http://ptolemy.eecs.berkeley.edu/publications`.

[68] Xi, H., and Pfenning, F. (1998). "Eliminating array bound checking through dependent types." *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI '98*), Montreal, pp. 249–257.

[69] Lee, E. A., and Xiong, Y. (2001). "System-level types for component-based design." *Proceedings of EMSOFT 2001*, Tahoe City, CA, Lecture Notes on Computer Science 2211, pp. 237–253. Springer-Verlag, Berlin.

[70] Lucassen, J. M., and Gifford, D. K. (1988). "Polymorphic effect systems." *Proceedings 15th ACM Symposium on Principles of Programming Languages*, pp. 47–57.

[71] Martin-Löf, P. (1980). "Constructive mathematics and computer programming." *Logic, Methodology, and Philosophy of Science VI*, pp. 153–175. North-Holland, Amsterdam.

[72] Hayashi, S. (1991). "Singleton, union, and intersection types for program extraction." *Proceedings of the International Conference on Theoretical Aspects of Computer Science* (A. R. Meyer, Ed.), pp. 701–730.

[73] Ullman, J. D. (1994). *Elements of ML Programming*. Prentice-Hall, Englewood Cliffs, NJ.

[74] Cardelli, L., and Wegner, P. (1985). "On understanding types, data abstraction, and polymorphism." *ACM Computing Surveys*, **17**, 471–522.

[75] Laddaga, R. (1998). "Active software." Position paper for the *St. Thomas Workshop on Software Behavior Description.*

[76] Gordon, M. J., Milner, R., Morris, L., Newey, M., and Wadsworth, C. P. (1978). "A metalanguage for interactive proof in LCF." *Conference Record of the 5th Annual ACM Symposium. on Principles of Programming Languages*, pp. 119–130. Assoc. Comput. Mach., New York.

[77] Wikstrom, Å. (1988). *Standard ML.* Prentice-Hall, Englewood Cliffs, NJ.

[78] Hudak, P. (1989). "Conception, evolution, and application of functional programming languages." *ACM Computing Surveys*, **21**, 359–411.

[79] Scott, D. (1970). "Outline of a mathematical theory of computation." *Proceedings of the 4th Annual Princeton Conference on Information Sciences and Systems*, pp. 169–176.

[80] Xiong, Y., and Lee, E. A. (2000). "An extensible type system for component-based design." *6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Lecture Notes on Computer Science, 1785, pp. 20–37. Springer-Verlag, Berlin.

[81] Rehof, J., and Mogensen, T. (1996). "Tractable constraints in finite semilattices." *Third International Static Analysis Symposium*, Lecture Notes in Computer Science 1145, pp. 285–301, Springer-Verlag, Berlin.

[82] von der Beeck, M. (1994). "A comparison of statecharts variants." *Proceedings of Formal Techniques in Real Time and Fault Tolerant Systems*, Lecture Notes on Computer Science 863, pp. 128–148. Springer-Verlag, Berlin.

[83] Johnson, R. E. (1997). "Frameworks = (Components + Patterns)." *Communications of the ACM*, **40**, 39–42.

[84] Abu-Ghazaleh, N., Alexander, P., Dieckman, D., Murali, R., and Penix, J. (1998). "Orbit—A framework for high assurance system design and analysis," TR 211/01/98/ECECS. University of Cincinnati.

[85] Alexander, P. (1998). "Multi-facetted design: The key to systems engineering." *Proceedings of Forum on Design Languages* (FDL-98).

[86] Jourdan, M., Lagnier, F., Maraninchi, F., and Raymond, P. (1994). "A multiparadigm language for reactive systems." *Proceedings of the 1994 International Conference on Computer Languages*, Toulouse, France.

[87] Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, A., and Trakhtenbrot, M. (1990). "STATEMATE: A working environment for the development of complex reactive systems." *IEEE Transactions on Software Engineering*, **16**, 403–414.