What's Ahead for Embedded Software?

Edward A. Lee
University of California, Berkeley

Once deemed too small and retro for research, embedded software has grown complex and pervasive enough to attract the attention of computer scientists. The most pressing problem is how to adapt existing software techniques to meet the challenges of the physical world.

Most of today's gadgets and cars use embedded software, which, in many cases, has taken over what mechanical and dedicated electronic systems used to do. Indeed, embedded software appears in everything from telephones and pagers to systems for medical diagnostics, climate control, and manufacturing. Its main task is to engage the physical world, interacting directly with sensors and actuators.

Research computer scientists have largely ignored embedded software because it has not been sufficiently complex or general to warrant the effort. The "Why Embedded Software Research Now?" sidebar describes how this is changing. There are many research questions, but most center around one issue: how to reconcile a set of domain-specific requirements with the demands of interaction in the physical world. How do you adapt software abstractions designed merely to transform data to meet requirements like real-time constraints, concurrency, and stringent safety considerations? The answer to this question has given rise to some promising research angles, including novel ways to deal with concurrency and real time and methods for augmenting component interfaces to promote safety and adaptability.

FRAMEWORKS

A framework is a set of constraints on components and their interaction, and a set of benefits that derive from those constraints. A framework defines a model of computation, which governs the interaction of components. Thus, the first step in understanding suitable models of computation is to understand what makes a framework useful for embedded system design. The embedded systems community must be very open about what "components" and "frameworks" might entail. Otherwise, we have little hope of getting a useful model because the prevailing component architectures in software engineering are not suitable for embedded systems.

Most frameworks have four service categories:

- *Ontology.* A framework defines what it means to be a component. Is a component a subroutine? A state transformation? A process? An object? An aggregate of components may or may not be a component. Certain semantic properties of components also flow from the definition. Is a component active or passive—can it autonomously initiate interactions with other components or does it simply react to stimulus?
- *Epistemology.* A framework defines states of knowledge. What does the framework know about the components? What do components know about one another? Can components interrogate one another to obtain information (that is, is there reflection or introspection)? What do components know

0018-9162/00/$10.00 © 2000 IEEE

Why Embedded Software Research Now?

Until recently, computer scientists have largely ignored embedded software. As a software problem, it was small, too retro in its use of quaint techniques such as assembly language programming, and too limited by hardware costs. The best software technologies, with their profligate use of memory, layers of abstraction, elaborate algorithms, and statistical optimization did not seem applicable. Because the research results did not fit the problem, the problem was not interesting.

This has changed for many reasons, and researchers are beginning to retool their research to address the very real and very different problems embedded software poses.

Hardware capabilities have improved. The techniques of old, such as automatic memory management or late binding to support polymorphism, seem within reach for embedded systems, but the techniques need significant adapting.

Design challenges

Embedded software is harder to design. Embedded systems are increasingly networked, which introduces significant complications such as downloadable modules that dynamically reconfigure the system. Moreover, consumers demand ever more elaborate functionality, which greatly increases software complexity. These systems can no longer be designed by a single engineer fine-tuning tens of kilobytes of assembly code.

Embedded software often encapsulates domain expertise, particularly when it must process sensor data or control actuators. Even very small programs may contain highly sophisticated algorithms, requiring a deep understanding of the domain and of supporting technologies, such as signal processing.

The emerging embedded software components business is a consequence of this complexity. It is very difficult to replicate a toll-quality speech coder or a radio modem with commodity programmers.

Existing software design techniques aren't suitable. Partly because it is recent, and partly because of the domain expertise it requires, embedded software is often designed by engineers who are classically trained in the domain, for example in internal combustion engines. They have little background in the theory of computation, concurrency, object-oriented design, operating systems, and semantics. In fact, it is arguable that other engineering disciplines have little to offer to the embedded system designer today because of their mismatched assumptions about the role of time and because of their profligate use of hardware resources. But these disciplines will be essential if embedded software is to become more complex, modular, adaptive, and network aware.

Interfacing to the real world

The drastic mismatch between many of the modern software techniques and the needs of embedded systems is not surprising if you remember that interfacing to the real world has only just begun to extend beyond keyboards and screens (which themselves are a relatively recent design emphasis). Computation has its roots in the transformation of data, not in the interaction with sensors, actuators, or even humans. Software in networked embedded systems, in contrast, will almost certainly be composed of components that operate concurrently and in real time, often interacting remotely.

about time? More generally, what information do components share? Scoping rules are part of the epistemology of many frameworks. Connectivity of distributed components, via name servers for example, is another part of the epistemology.

- *Protocols.* A framework provides mechanisms that dictate how components interact. Do they use asynchronous message passing? Rendezvous? Semaphores? Monitors? Publish and subscribe? Timed events? Sequential transfer of control?
- *Lexicon.* This is the vocabulary of component interaction. For components that interact by sending messages, the lexicon is a type system that defines the possible messages. The words of the vocabulary are types in some languages (or family of languages, as in CORBA).

Along any of these dimensions, a framework 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. The Internet is a framework that primarily constrains the lexicon (byte streams) and the protocols (TCP/IP, UDP, and HTTP). These constraints produce the primary benefit of platform independence.

A framework is often deeply ingrained in the culture of the designers who use it. Consequently, designers fail to consider concepts and ideas that are meaningful but outside that culture. The framework essentially fades into the background of a particular application area. The Turing sequentiality of computation, for example, is so deeply ingrained in contemporary computer science culture that we no longer realize just how thoroughly we have banished time from the domain of discourse. Common practice in concurrent programming is that the framework components are threads (the ontology), which use semaphores and monitors (the protocols) to share memory (the epistemology) and exchange objects (the lexicon). This is a very broad framework with few benefits. It is particularly hard to talk about the properties of an aggregate of components because the aggregate no longer has the properties of its individual components. Indeed, it is very difficult to characterize the aggregate component at all, as the "A Broader View of Components and Frameworks" sidebar describes.

The key challenge in embedded software research then is to invent frameworks with properties that better match the application domain. One requirement is to reintroduce time. Another is to recognize that certain essential properties (safety and liveness, for example) compose when components become an aggregate.

Concurrency

A framework that allows concurrency is particularly useful in designing embedded systems. A model in such a framework consists of components that can perform

A Broader View of Components and Frameworks

The terms "component" and "framework" are overused. Sometimes the meaning is very specific—components are "distributed objects." Sometimes it is very broad—components are any kind of building block. I prefer the very broad interpretation, because none of the established narrow interpretations match the needs of embedded software. Designers construct complex embedded software from distinct modules of some sort. The modules are the components, and the framework is the mechanism by which the modules interact. Ideally, the modules will be reusable and embody valuable domain-specific expertise.

Reusable software components for embedded systems are already a viable business, particularly in signal processing, where considerable domain expertise is encapsulated. However, the definition of these components is ad hoc, the framework is unsophisticated, and their role in the embedded system is totally static. As embedded systems become network aware and configurable, these traditional components will not adapt well. Networked embedded systems are likely to alter their architecture dynamically, as agents, changing service demands, and new components arrive over the network.

Subroutines

The most widely applied software component technology is probably subroutines—finite computations that take predefined arguments and produce final results. Subroutine libraries are marketable component repositories, but they 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. You can do it, of course, particularly with the help of syntactic mechanisms such as objects, which make it easier to package subroutines with data that persists across those subroutines' calls. However, a speech coder is more like a process than a subroutine. It is a nonterminating computation that transforms an unbounded stream of input data into an unbounded stream of output data. A commercial speech coder for cellular telephony is likely to be defined as a process that expects to execute on a dedicated signal processor.

Processes and threads

Processes and their cousins, threads, are widely used for concurrent software design. Indeed, you can view processes as a component technology, in which a multitasking operating system or multithreaded execution engine provides the framework that coordinates the components. The framework supports component interaction mechanisms, such as monitors, semaphores, and remote procedure calls. In this context, a process is a component that exposes at its interface an ordered sequence of external interactions. As a component technology, however, processes and threads are extremely weak. From an external view, a process or thread is a sequence of interactions. An aggregate of two processes, however, does not constitute a process because it no longer exposes an ordered sequence of external interactions. Indeed, without considerable extra effort, probably using semaphores and monitors, the external interactions no longer have a well-defined order. Without imposing additional constraints, two processes do not form any component that you can easily (and usefully) characterize—which is why concurrent programs built from processes or threads are so hard to get right. It is very difficult to talk about the aggregate's properties because you don't know what the aggregate is.

Semaphores and monitors are at the assembly-language level of concurrency and are too difficult for anyone but operating system experts to use reliably. (Most engineers completely understand only trivial designs.) Overly conservative rules of thumb dominate, such as "Always grab locks in the same order."[1] Concurrency theory has much more to offer than concurrency practice, but again it probably needs adapting for embedded system design. For example, it often reduces concurrency to interleavings, which trivializes time by asserting that all computations are equivalent to sequences of discrete timeless operations.

Frameworks

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 definition is broader than (but consistent with) the framework definition in object-oriented design.[2] By this definition, there are many, many frameworks, some of which are purely conceptual, cultural, or even philosophical. Others are embodied in the software. Operating systems are frameworks in which the components are single programs or processes. Programming languages are frameworks in which the components are language primitives and aggregates of these primitives, and where the grammar defines the possible interactions. Distributed component middleware such as the Common Object Request Broker Architecture (CORBA) and Distributed Component Object Model (DCOM) are frameworks. Synchronous digital hardware design principles are a framework. JavaBeans form a framework that is tuned to user interface construction. A particular class library and policies for its use is a framework.

As is true of all applications—not just embedded systems—the choice of framework must fit the application domain. In embedded systems, this means choosing frameworks that use concurrency and can deal with real-time constraints. Operating systems with no real-time facilities won't be that useful, for example (see Table 1 in the main text for a list of frameworks that may be suitable).

References

1. D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, Reading, Mass., 1997.
2. R.E. Johnson, "Frameworks = (Components + Patterns)," *Comm. ACM*, Oct. 1997, pp. 39-42.

Table 1. Sample frameworks useful in embedded software design.

Framework	Component interaction mechanism	Possible applications
JavaBeans, COM, and CORBA	Unstructured events, no built-in synchronization	Good foundation for more disciplined models (such as CORBA's event service). Because no synchronization is built in, programmers can easily implement unsynchronized interactions with no risk of deadlock. However, if synchronization is required, for example to enforce data precedence, the programmer must build up the mechanisms from scratch. It then becomes difficult to maintain determinacy and avoid deadlock.
Publish and subscribe	Event notification	A component declares an interest in a family of events (subscribes); another component asserts events (publishes). Some of the more sophisticated realizations are based on Linda, for example JavaSpaces from Sun Microsystems. A similar realization is in CORBA's event service (see N. Carriero and D. Gelernter, "Linda in Context," *Comm. ACM*, Apr. 1989, pp. 444-458).
Asynchronous message passing	Processes send messages through channels that can buffer the messages.	Several variants exist, some with strong formal properties, such as Kahn process networks and dataflow models.
Synchronous message passing	Processes rendezvous, communicate in atomic instantaneous actions	Hoare's Communicating Sequential Processes (CSP) and Milner's Calculus for Communicating Systems (CCS). Underlies several concurrent languages, including Lotos and Occam. Timed extensions are particularly pertinent.
Discrete events	Components communicate via signals that carry events placed in time, which is globally known (by all components)	Hardware description languages, including VHDL and Verilog, as well as several modeling languages, for example for communication networks.
Synchronous/ reactive	Global clock triggers computations that are conceptually simultaneous and instantaneous. Signals consist of data values that are aligned with global clock ticks.	Esterel, Signal, and Lustre (see A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems," *Proc. IEEE*, IEEE Press, Piscataway, N.J., Vol. 79, No. 9, 1991, pp. 1270-1282).
Discrete-time	Same as synchronous/reactive model, except every signal has a value every clock tick.	Cycle-driven frameworks, such as that embodied in SystemC (http://www.systemc.org), although with embellishments they look a bit like discrete-time models.
Continuous-time	Processes communicate via continuous-time signals, which are functions on the real numbers.	Includes differential equations as used in Simulink, Saber, and VHDL-AMS, all of which model the physical world.

some computation in parallel, at least conceptually (and perhaps actually). A concurrent framework defines the "laws of physics," fashioning the ontology, protocols, and epistemology to achieve a particular approach to concurrent computing. In practice, concurrency seriously complicates system design. No universal concurrent framework has yet emerged (despite what some proponents of a particular approach might say).

To understand why, consider the von Neumann framework, a universally accepted model of sequential computation. A key part of its success is that it reduces time to a total order of discrete events, in which sequencing is sufficient for correctness. In distributed systems, maintaining such a total order globally is expensive, except for very small systems. Thus, in practice, events are partially ordered at best. This partial ordering makes it difficult to maintain a global notion of "system state," which is an essential part of the von Neumann framework.

In networked embedded systems, communication bandwidth and latencies will vary over several orders of magnitude, even within the same system design. A framework well suited to small latencies (the synchronous hypothesis in digital circuit design, for example, 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.

Gul Agha of the University of Illinois describes *actors*, which extend objects to concurrent computation.[1] Actors encapsulate a thread of control and have interfaces for interacting with other actors. The protocols for this interface, *interaction patterns*, are part of the concurrent framework. Agha argues that no model of concurrency can or should let designers express all communication abstractions directly. He describes message passing as akin to `gotos` in their lack of structure. Instead, he recommends using an interaction policy to determine how actors are composed.

Sample frameworks

Both researchers and practitioners have explored a rich variety of frameworks that deal with concurrency and time in different ways. Table 1 lists a few examples. So far, most designers are exposed to only one or two frameworks, but as design practices change—

Functionality has steadily shifted from hardware to software.

as the level of abstraction and domain-specificity rise—this diversity will make it hard to select a framework. Embedded system designers will soon need some way to reconcile the myriad views being offered.

Time is one dimension in which these views differ. Some models of computation are very explicit: They view time as a real number that advances uniformly, and they place events on a timeline or evolve continuous signals along the timeline. Others are more abstract, viewing time as discrete. Still others are even more abstract, viewing time as merely a constraint imposed by causality. In this last interpretation, time is partially ordered. The rich mathematics of partial orders then provides a mathematical framework for formally analyzing and comparing computational models.[2] Process networks and rendezvous-based models take this view, which explains much of their expressiveness.

Many researchers have thought deeply about the role of time in computation. Albert Benveniste and colleagues at the Institut Recherche en Informatique et Systems Aleatoires (IRISA) 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."[3] Leslie Lamport of the Compaq Systems Research Center observes that there is no way to exactly maintain a coordinated notion of time in distributed systems and shows that a partial ordering is sufficient.[4]

Mixing frameworks

A grand unified approach to modeling would seek a concurrent framework that serves all purposes. One approach is to create the union of all the frameworks, providing all of their services in one bundle. But the resulting framework would be extremely complex and difficult to use, and designing synthesis and validation tools would be difficult. A more feasible alternative is to choose one concurrent framework, say rendezvous, and show that all the others are special cases of that.

This is relatively easy to do—in theory. Most of these frameworks are sufficiently expressive to subsume most of the others. The disadvantage is that this approach doesn't acknowledge each model's strengths and weaknesses.

Another approach is to use an architecture description language (ADL) to define a framework. Some ADLs, such as Wright,[5] can describe the rich component interactions common in software architecture and often provide a way to get good insights into the design. But sometimes the ADL and the design are a poor match. Wright, for example, which is based on CSP, does not cleanly describe asynchronous message passing (it requires overly detailed descriptions of the message-passing mechanisms).

What we really need are architecture *design* languages, not architecture *description* languages. That is, the focus should not be on describing current practice, which an ADL does, but on improving future practice. Wright, with its strong commitment to CSP, should not be concerned with whether it cleanly models asynchronous message passing. It should take the stand that asynchronous message passing is a bad idea for the designs it does describe well.

A final alternative is to heterogeneously mix frameworks, but instead of forming the union of their services, preserve their distinct identity. A few mixtures combine concurrency models with sequential models such as finite-state machines. For example, hybrid systems[6] combine finite-state machines with continuous-time models. Statecharts and variants combine synchronous/reactive models with finite-state machines. "*charts" (pronounced "starcharts") combine finite-state machines with a variety of other concurrency models.[7] Yet another interesting integration of diverse semantic models is in Statemate,[8] which combines activity charts (a process network variant) with statecharts.

HARDWARE-SOFTWARE PARTNERSHIP

Since the 1970s, when programmable DSPs and microcontrollers first appeared, functionality has steadily shifted from hardware to software. This glib statement actually has profound consequences. "Software" means *primarily sequential* execution with a single instruction stream. That is, the same hardware resources are multiplexed in time to perform a variety of functions. "Hardware," in contrast, means *primarily parallel* execution; hardware resources are not shared (or at least, not as much). Of course, between these endpoints is a continuum: To some degree, software executes in parallel and hardware functional units are multiplexed.

Most embedded systems involve both significant hardware and software design, so a large part of a designer's task is to explore the balance between their sequential and parallel execution styles. For hard-real-time functions, such as signal processing, designers often assign concurrent tasks to distinct processors. For example, the speech coders and radio modems in a digital cellular telephone use processors distinct from the microcontroller that handles the overall control logic. Thus, despite being primarily software components, the speech coders and radio modems have a hardware nature in that they require dedicated hardware resources that are not multiplexed.

In theory, as embedded processor performance improves, there should be less need for such hardware specialization. Until then, however, designers must use dedicated hardware to handle hard-real-time tasks or use processors that so greatly exceed minimum performance capabilities that failure is unlikely despite the unpredictability that multitasking introduces. Real-time

operating systems cannot yet reliably handle many hard-real-time tasks, and before this can change, the embedded system community must rethink multitasking. First, component interface definitions need to declare temporal properties, not just a fixed priority, which is sufficient only under the rarely applicable assumptions of rate-monotonic scheduling. The definitions need to declare the dynamics (phases of execution, exception handling, modes of operation, and yes, also periodicity, where appropriate). Second, compositions of components must have consistent and nonconflicting temporal properties—much as compositions of objects must have compatible types where they interact. One possible approach, which synchronous/reactive languages take, for example, views all executing processes as part of a single application. Programmers build programs in a highly concurrent, hardware-like manner, but then compile away the concurrency.

REAL-TIME SCHEDULING

Although scheduling is an old topic, it has certainly not played out. A real-time scheduler provides some assurances of timely performance given certain component properties, such as a component's invocation period or task deadlines. Rate-monotonic scheduling principles translate the invocation period into priorities. Priorities may also be based on semantic information about the application, reflecting the criticality with which the scheduler must deal with some event, for example.

Unfortunately, most methods are not compositional. Even if a method can provide assurances individually to each component in a pair, there is no systematic way to provide assurances for the aggregate of the two, except in trivial cases. One manifestation of this problem is priority inversion—a chronic issue in scheduling. Priority inversion occurs when processes interact, for example by entering a monitor to exclusively access a shared resource. Suppose a low-priority process accesses and locks a shared resource, and then a medium-priority process preempts it. The low-priority process holds exclusive access to the resource, but cannot execute while the medium priority process is around. Suppose that a high-priority process preempts the medium-priority process, and then attempts to gain access to the same resource. It must wait for the medium-priority process to run to completion, and then for the low-priority process to run until it relinquishes the resource. In effect, the low-priority process blocks the high-priority process.

Although there are ways to prevent priority inversion, 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) that the framework supports. Together, these interaction mechanisms have no coherent compositional semantics, which points to the need for a different scheduling mechanism entirely. This could be a fruitful research area.

INTERFACES AND TYPES

Type systems are one of the great practical triumphs of contemporary software. They do more than any other formal method to ensure software's correctness. Object-oriented languages, with their user-defined abstract data types, have greatly enhanced both software reusability (witness the Java class libraries) and software quality. Type systems provide a vocabulary for talking about larger structure than lines of code and subroutines.

Type systems are one of the great practical triumphs of contemporary software. They do more than any other formal method to ensure software's correctness.

The disadvantage for embedded software is that type systems talk only about static structure—the *syntax* of procedural programs. They say nothing about the program's concurrency or dynamics. These properties are relegated to more informal descriptions, such as design patterns and object modeling. For example, calling the initialize() method before the go() method is not part of the object's type signature. An object's temporal properties (method x() must be invoked every 10 ms) are also not part of the type signature. Work with active objects and actors moves a bit in the right direction by being somewhat more explicit about the dynamic properties of the components' interfaces, but it does not say enough about interfaces to ensure safety, liveness, consistency, or real-time behavior. Design by contract moves in the right direction, but it has weak formal properties.

Type system techniques

At its root, a type system constrains what a component can say about its interface and how to ensure compatibility when designers compose components. Mathematically, type system techniques depend on a partial type ordering, typically defined by a subtyping relation or (more ad hoc) by lossless convertibility. This means that an object of type *A* can be converted to an object of type *B* without any loss of information, as Figure 1 illustrates. Designers can build elaborate techniques from the robust mathematics of partial orders, leveraging fixed-point theorems, for example, to ensure convergence of type checking, type resolution, and type inference algorithms. You can even view type system methods as special cases of theorem-proving methods.

With this very broad interpretation of type systems, embedded software designers can use the theory as long as the interface properties are elements of a partial order—preferably a complete partial order or a lattice.[9] I suggest they first describe an interface's dynamic properties (such as the protocols with which a component interacts with other components) using nondeterministic automata. They can then define a

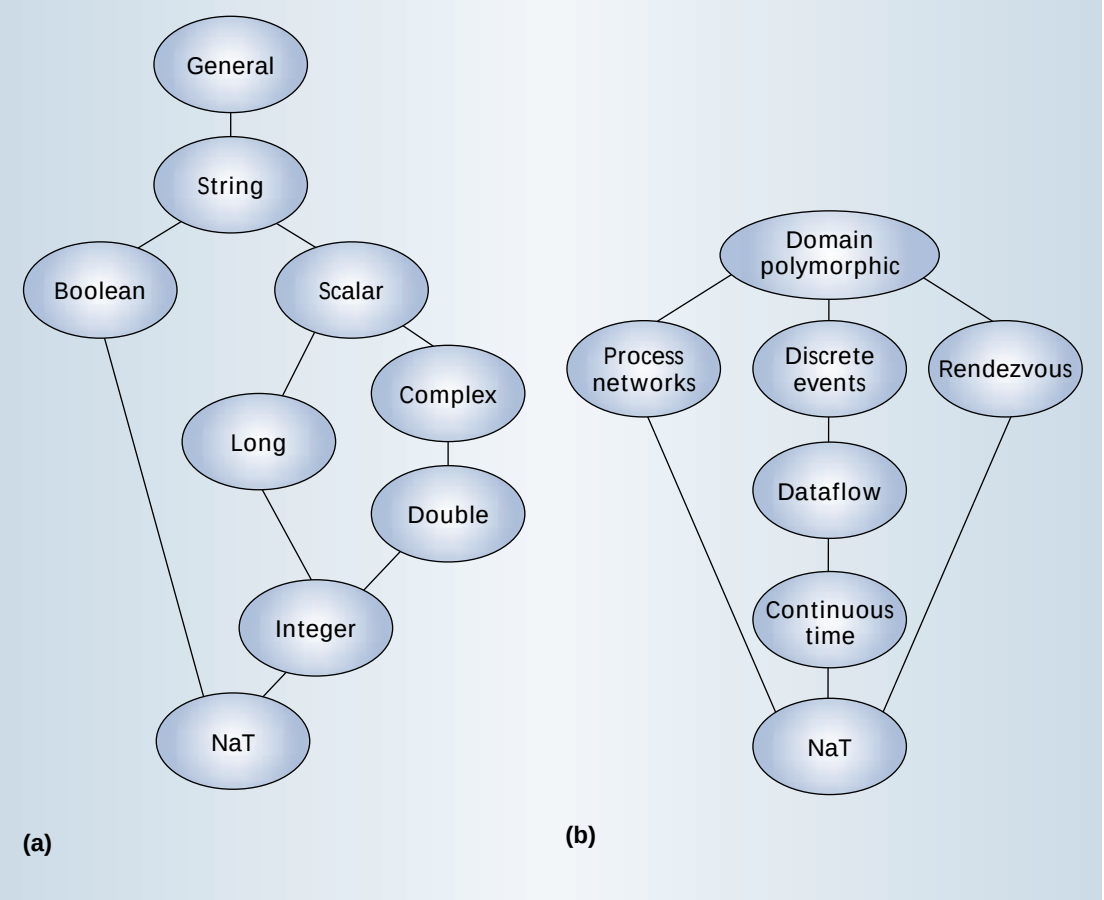

Figure 1. How a type system works. A type system makes it possible to ensure compatibility when composing components. Partial type ordering is defined by lossless convertibility. In (a), a data-level type system, a data type is "less than" another type if it can be converted to the other type without loss of information. For example, Integer is less than Double and less than Long, but Double and Long are incomparable (neither is less than the other). In (b), a system-level type is less than another if the other simulates the first. System-level types use automata to capture the dynamics of component interaction. (This idea is explained in detail at http://ptolemy.eecs.berkeley.edu/publications/papers/00/systemLevel.)

pertinent partial ordering relation using the simulation relation among automata. It may also be possible to use various timed automata extensions in similar ways to define an interface's temporal properties much more completely than is possible in common practice today.

The case for strong typing

Strongly typed languages, such as Java and ML, emphasize catching errors as soon as possible—often the compiler catches them. But many errors are not within the type system's scope. Accessing an array out of bounds, for example, is a typical programming error that the type system cannot prevent. Several researchers have shown how to augment the type system to include such properties as array size (typically using dependent types[10]). This work lends credence to the idea of extending type systems to program dynamics.

Not everyone is enthusiastic about strong typing, however. John Ousterhout of Scriptics Inc. argues that strong typing compromises modularity and discourages reuse:[11]

> Typing encourages programmers to create a variety of incompatible interfaces...each interface requires objects of specific type and the compiler prevents any other types of objects from being used with the interface, even if that would be useful.

Instead, Ousterhout advocates languages without strong typing, such as Lisp and Tcl, where safety is possible only with extensive runtime checking. Type checking is postponed until the last possible moment, however, so a system may exhibit erroneous behavior only after running for an extended period after the actual violation. Identifying the source of the problem can be difficult, and guaranteeing the code may be impossible.

Although Ousterhout raises a valid point, discarding strong typing is not the solution. For embedded systems especially, the extra degree of safety that strong typing offers overwhelms even the desire for modularity and reuse. The question then becomes how to achieve modularity and reuse without discarding strong typing.

One solution is to use polymorphism, reflection, and runtime type inference and type checking. Components must give their dynamic properties as part of their interface definition. Automata can give the protocols for communication among components, for example.

Subclassing and polymorphism become possible because one automaton can be a generalization of another (technically, via a simulation relation), much the way a complex number data type is a generalization of a real-number data type.

Extending types to include an interface's dynamic properties requires novel syntactic language support as well as new compiler and runtime techniques. Object-oriented languages, for example, brought typing concepts such as inheritance and polymorphism into mainstream use. Not only do modern languages syntactically support these concepts, but higher level visual syntaxes such as the Unified Modeling Language (UML) have evolved to extend syntactic support even further. Extended type systems could, in principle, reflect

- protocols for communication between components (rendezvous, asynchronous message passing, streams, events, and so on);
- models of time (continuum, discrete, clocked, partially ordered, and so on); and
- control flow (synchronous, scheduled firings, process scheduling, real-time, and so on).

These component aspects can be polymorphic, meaning that components will assert minimal constraints in their interface. To make such types possible, there must be sufficient syntactic language support—a key part of future embedded software research.

METAFRAMEWORK

All frameworks impose some constraints to achieve certain benefits. As a rule, stronger benefits come at the expense of stronger constraints. Thus, frameworks can 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 complex systems will have to mix frameworks heterogeneously. There are several ways to do this. One is through specialization (analogous to subtyping) where one framework is simply a more restricted version of another. A second way is to mix frameworks hierarchically. A component in one framework is actually an aggregate of components in another. The challenge is to avoid having to design each pairwise hierarchical framework combination.

The Ptolemy project at UC Berkeley (http://ptolemy.eecs.berkeley.edu) takes the hierarchical approach and uses a system-level type concept, *domain polymorphism*, to avoid pairwise design. Designers realize a framework using a software infrastructure, called a *domain*. A component that is domain polymorphic can operate in multiple domains. The idea is that the interface an aggregate of components exposes in a domain is itself domain polymorphic. Thus, designers can use the aggregate in any of several other domains and still have clear semantics.

Initially, Ptolemy project members built domain polymorphic components in an ad hoc way, using intuition to define an interface that was as unspecific as possible. More recently, they have characterized these interfaces using nondeterministic automata to give the interface's assumptions and requirements precisely. Automata also characterize the services each domain provides. A component can operate within a domain if its interface automata simulate those of the domain. The resulting Ptolemy framework can be viewed as a metaframework in that it provides an infrastructure for composing frameworks.

All frameworks impose some constraints to achieve certain benefits. As a rule, stronger benefits come at the expense of stronger constraints.

A few other research projects have also combined computational models hierarchically. The Gravity system and its visual editor Orbit, like Ptolemy, provide a metaframework that mixes modeling techniques heterogeneously.[12] A model in a domain is a *facet*, and heterogeneous models are *multifaceted designs*.

These are but a few of the interesting embedded system research problems. There are many more. Human-computer interaction, for example, is key to making embedded systems pervasive and useful. Ideally, the embedded software becomes transparent, mediating a natural and intuitive interaction with the physical world. Also, configurable hardware offers interesting opportunities and challenges and potentially relates strongly to the problem of selecting appropriate computational models.

There are also interesting and challenging networking problems, particularly providing quality-of-service guarantees in the face of unreliable resources. Finally, hardware and software design techniques that minimize power consumption are critical for portable devices.

I have focused on constructing embedded software because embedded software demands that time become a first-class part of the programming exercise. Embedded system designers need more than threads, semaphores, and monitors. The focus must move beyond a program's functional correctness to its temporal correctness. The key problem then becomes identifying the appropriate abstractions for representing the design. ✸

Acknowledgments

The character and role of frameworks became much clearer to me after I attended the Workshop on Software Behavior Description (December 1998) and the Workshop on Software Development for the Post-PC World (December 1999). I thank Cordell Green,

Tom Henzinger, Paul Hudak, Gregor Kiczales, Bob Laddaga, Butler Lampson, Bill Mark, John Mitchell, Bill Scherlis, Victoria Stavridou, and Janos Sztipanovits. I drew many technical ideas from the Ptolemy project and have been particularly influenced by many of the students and postdocs in that project. I also thank Kees Vissers for extensive, thoughtful feedback.

References

1. G.A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, Mass., 1986.
2. E.A. Lee and A. Sangiovanni-Vincentelli, "A Framework for Comparing Models of Computation," *IEEE Trans. CAD Integrated Circuits and Systems*, Dec. 1998, pp. 1217-1229.
3. A. Benveniste and P. Le Guernic, "Hybrid Dynamical Systems Theory and the SIGNAL Language," *IEEE Trans. Automatic Control*, May 1990, pp. 525-546.
4. L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM*, July 1978, pp. 558-565.
5. R. Allen and D. Garlan, "Formalizing Architectural Connection," *Proc. 16th Int'l Conf. Software Eng.* (ICSE 94), IEEE CS Press, Los Alamitos, Calif., 1994, pp. 71-80.
6. T.A. Henzinger, "The Theory of Hybrid Automata," *Proc. 11th Symp. Logic in Computer Science*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 278-292.
7. A. Girault, B. Lee, and E.A. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models," *IEEE Trans. CAD Integrated Circuits and Systems*, June 1999, pp. 742-760.
8. D. Harel et al., "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Trans. Software Eng.*, Apr. 1990, pp. 403-414.
9. W.T. Trotter, *Combinatorics and Partially Ordered Sets*, Johns Hopkins Univ. Press, Baltimore, 1992.
10. P. Martin-Löf, "Constructive Mathematics and Computer Programming," in *Logic, Methodology, and Philosophy of Science VI*, North-Holland, Amsterdam, 1980, pp. 153-175.
11. J.K. Ousterhout, "Scripting: Higher Level Programming for the 21st Century," *Computer*, Mar. 1998, pp. 22-30.
12. N. Abu-Ghazaleh et al., "Orbit—A Framework for High Assurance System Design and Analysis," Tech. Report TR 211/01/98/ECECS, Univ. of Cincinnati, 1998.

Edward A. Lee is a professor of electrical engineering and computer sciences at the University of California, Berkeley, where his research interests center on design, modeling, and simulation of embedded, real-time computational systems. He received a PhD in electrical engineering from UC Berkeley. His addresses are eal@eecs.berkeley.edu and http://ptolemy.eecs.berkeley.edu/~eal.