EMBEDDED SOFTWARE --
AN AGENDA FOR RESEARCH

by

Edward A. Lee

Memorandum No. UCB/ERL M99/63

11 December 1999

COVER

EMBEDDED SOFTWARE --
AN AGENDA FOR RESEARCH

by

Edward A. Lee

Memorandum No. UCB/ERL M99/63

11 December 1999

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Embedded Software — An Agenda for Research

Edward A. Lee

University of California at Berkeley
eal@eecs.berkeley.edu

December 11, 1999

1.0 EMBEDDED SOFTWARE

Embedded software is that which resides in devices that are not first-and-foremost computers. It is pervasive, appearing in vehicles, telephones, pagers, audio equipment, aircraft, appliances, toys, security systems, games, PDAs, medical diagnostics, weapons, pacemakers, television sets, video production equipment, network switches, printers, scanners, climate control systems, manufacturing systems, etc. Embedded software engages the physical world, interacting directly with sensors and actuators.

A technically active person today probably interacts regularly with more embedded software than conventional programs. This is a relatively recent phenomenon. Not so long ago automobiles depended on finely tuned mechanical systems for the timing of ignition and its synchronization with other actions. Today, embedded software has taken over these functions.

Embedded software is traditionally the domain of practicing engineers, not research scientists. As a software problem, it has been viewed by some as too 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 just did not seem applicable. Since the research results did not fit the problem, the problem was not interesting.

This has changed. Researchers now recognize the importance of the area and are beginning to retool their research to address the very real, and very different software problems of embedded systems.

One reason for the change is simply that hardware capabilities have improved sufficiently that the profligate techniques of old seem within reach for embedded systems. However, this proves only an enticement to look at the problem. The techniques need significant adaptation.

A second reason for the change is that embedded software has become much 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 the complexity of the software, making it much more difficult for a single engineer to accomplish the task by finely tuning a few tens of kilobytes of assembly code.

1.1 Components

Components are an old concept with a new urgency. It seems obvious that complex embedded software will have to be constructed from distinct modules of some sort. Ideally, these modules are reusable, and embody valuable expertise in one or more aspects of the problem domain.

Software components for embedded systems are already a viable business. Many modems and cellular telephones incorporate software components licensed from third parties (primarily for the signal processing functions such as speech coding and the radio modem). While these components do embody considerable expertise with some very difficult functionality, their definition is ad-hoc, their architecture unsophisticated, and their role in a system totally static. They will not adapt well as these systems are made network aware and configurable. Networked embedded systems are likely to dynamically alter their architecture, as agents, changing service demands, and new software components arrive over the network.

In software, it is arguable that the most widely applied component technology is subroutines. Subroutines are finite computations that take pre-defined arguments and produce final results. Subroutine libraries are marketable component repositories.

Subroutines, 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, particularly with the help of syntactic mechanism such as objects, which facilitate packaging data that persists across subroutine calls together with those subroutines. However, a speech coder is more like a process than a subroutine. It is a non terminating 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.

Processes, and their cousin, threads, are widely used for concurrent software design. Indeed, processes can be viewed as a component technology, where a multitasking operating system or multithreaded execution engine provides the framework that coordinates the components. Component interaction mechanisms, monitors, semaphores, and remote procedure calls, are supported by the framework. In this con-

text, 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.

1.2 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[1]. This is broader than, but consistent with the definition of frameworks in object-oriented design [43]. 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 [9] 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 [43].

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 evaluate the usefulness of a framework, it is helpful to orthogonalize its services.

- **Ontology**. A framework defines what it means to be a component. For example, is a component a subroutine? A state transformation? A process? An object? A consequence of this definition is that 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 (i.e. is there reflection or introspection)? What do components know about time? More generally, what information do components share? Scoping rules are part of the epistemology of many frameworks. Connectivity of distributed components are another part of the epistemology.
- **Protocols**. A framework constrains the mechanisms by which components can interact. Do they use asynchronous message passing? Rendezvous? Semaphores? Monitors? Publish and subscribe? Timed events? Transfer of control? In the latter case, the components are states and their interactions are state transitions.
- **Lexicon**. The lexicon of a framework is the vocabulary of the interaction of components. 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 language (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.

Common practice in concurrent programming is that the framework components are threads (the ontology), which share memory (the epistemology), and exchange objects (the lexicon) using semaphores and monitors (the protocols). This is a very broad framework 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.

A framework is often deeply ingrained in the human culture of the designers that use the framework. It fades out of the domain of discourse. I will argue 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 the domain of discourse.

The key challenge in embedded software research is to invent frameworks with properties that better match the application domain. One of the requirements is that time be re-introduced.

1.3 Models of computation

A particularly useful sort of framework is a concurrent model of computation. The components in such a model are entities capable of performing some computation, where conceptually (and maybe actually) the components perform their computations in parallel. The model may restrict the components further by stating, for example, that components perform finite computations only in response to stimulus (from other components or from the framework, as for example when a run-time scheduler is used). The mechanism by which components communicate is defined by the model of computation. A model of computation, therefore, defines the ontology and protocols of a framework, and possibly some or all of the epistemology, but none of the lexicon. Later we will give many examples of concurrent models of computation.

1. This elegant definition is due to Bob Laddaga.

1.4 Domain specificity

Embedded software often encapsulates domain expertise, particularly when processing sensor data or controlling actuators. Even very small programs may contain very sophisticated algorithms, requiring deep understanding of the domain and of supporting technologies such as signal processing. The emerging embedded software components business is a consequence of this. It is very difficult to replicate a toll-quality speech coder or a radio modem with commodity programmers.

Partly because it is recent, and partly because of the domain expertise required, 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 these 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.

1.5 Computation

Why do we need to invent new frameworks? There are so many already. It is helpful to examine the underlying assumptions in modern approaches to computation to see what is missing.

- Time has been systematically removed from theories of computation, since it is an annoying property that computations take time. "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 have so little to go on that they 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 [64]. Attempts to imbue object-oriented design with real-time are far from satisfactory [22]. Variants with more direct temporal representations, such as timed I/O automata [63], represent a step in the right direction, and need to be adapted for the embedded system design culture. Even the widely used term "synchronous," despite the Greek root "chronos," has little to do with time (as usually used). It is a logical property of events, a set of ordering constraints. Embedded systems confront time the way humans do: as a relentless, measurable march.
- In software practice, management of concurrency is extremely primitive. Semaphores and monitors are the assembly language level of concurrency. They are too difficult to use reliably, except by operating system experts. Only trivial designs are completely comprehensible (to most engineers). Over conservative rules of thumb dominate (such as: always grab locks in the same order [53]). Concurrency theory has much more to offer than concurrency practice, but again it probably needs adaptation for the embedded system context. For instance, it often reduces concurrency to "interleavings," which trivialize time by asserting that all computations are equivalent to sequences of discrete time-less operations.
- In embedded systems, liveness is a critical issue. Programs must not terminate. In the Turing-Church view of computation, all non-terminating programs fall into an equivalence class, 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.
- Type systems are one of the great practical triumphs of contemporary software. They do more than any other formal method to ensure correctness of software. Object-oriented languages, with their user-defined abstract data types, have had a big impact in both reusability of software (witness the Java class libraries) and the quality of software. Combined with design patterns [26] and object modeling [24], type systems give us a vocabulary for talking about larger structure in software than lines of code and subroutines. 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 go() method. Temporal properties of an object (method x() must be invoked every 10ms) are also not part of the type signature. Work with active objects and actors [3][4] move a bit in the right direction by being a bit more explicit about dynamic properties of the interfaces of components. But they do not say enough about interfaces to ensure safety, liveness, consistency, or real-time behavior.
- Computer architecture, which would seem to be closer to the physical world, and hence closer in spirit to embedded systems, has been tending towards making things harder for the designers of embedded systems. Much 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 greatly compromise the reliability of embedded systems the way they are designed today, and in fact are mostly not used in embedded processors such as programmable DSPs and microcontrollers. I believe that these techniques have such a big impact on average case performance that they are indispensable. But the software world will have find abstractions that regain control of time, or the embedded system designers will continue to refuse to use these processors.

The drastic mismatch between many of the modern software techniques and the needs of embedded systems is not surprising in view of the fact that interfacing to the real world has only recently begun to extend beyond keyboards and screens. (We should not forget that even the emphasis on keyboards and screens is relatively recent.) Computation has its roots in the transformation of data, not in the interaction

with sensors, actuators, and humans.

1.6 Real time software

Starting when programmable DSPs [51] and microcontrollers appeared in the 1970s, functionality has been steadily shifting from hardware to software. This glib statement actually has profound consequences. What we mean by "software" is *primarily sequential* execution, where the same hardware resources are multiplexed in time to perform a variety of functions. That is, there is a single instruction stream. What we mean by "hardware" is *primarily parallel* execution, where hardware resources are not shared among functions (or at least, not as much). Of course, there is a continuum in between, with parallel execution of software and multiplexing of hardware function units. Most embedded systems have significant hardware design as well as significant software design, and a major part of the design space exploration considers the balance between these sequential and parallel execution styles.

The trend in embedded systems is certainly towards integrated processors that merge a variety of hard-real-time functions into a single instruction stream, towards the software end of the continuum (see for example [77]). However, much work remains to be done before a single instruction stream is a viable option for the high performance functions such as signal processing.

For hard-real-time functions, such as signal processing, concurrent tasks are often assigned distinct processors. For example the speech coders and radio modem operations 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, unmultiplexed hardware resources.

In theory, as the performance of embedded processors improves, there should be less need for such hardware specialization. However, real-time operating systems are not yet able to reliably handle many hard-real time tasks. In practice, they are handled today by either dedicated hardware or by processors that so greatly exceed the minimum performance capabilities that failure is unlikely despite the unpredictability introduced by multitasking.

Before this situation can change, we have to rethink multitasking. First, component interface definitions need to declare temporal properties. Not just a priority, which is only sufficient under the rarely applicable assumptions of rate-monotonic scheduling [60][48], but they need to declare the dynamics (phases of execution, exception handling, modes of operation, and yes, also periodicity where appropriate). Then compositions of components need to have consistent and non-conflicting temporal properties, much as today compositions of objects need to have compatible types where they interact.

One possible approach views all executing processes as part of a single application. Since processes can come and go, this single application has a dynamically changing software architecture. It may be better viewed as a dynamically changing application than as a set of disjoint applications with minimal interconnectedness. When components are composed, methods similar to the compile-time and run-time type validation of Java need to be applied.

1.7 Concurrency

Embedded systems engage the physical world. And in the physical world, 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 [10] such as Esterel [13], 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 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 difficult. 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.

1.8 Requirements

For embedded software, we need concurrent models of computation that are easier to get right than threads, semaphores, and monitors. We need to make time a first-class part of the programming exercise, and talk about the temporal correctness, not just functional correctness of programs.We need to discard termination as a criterion for correctness (and reuse it as a criterion for incorrectness).

This requires rethinking some concepts. The undecidability of halting 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. "Synchronization" needs to once again take into account "chronos." And correctness cannot be viewed as getting the right final answer. It has to take into account the timeliness of a continuing stream of partial answers.

Embedded software design has much in common with hardware design, so perhaps there are some lessons to be learned by looking at hardware design. Hardware is highly concurrent. Conceptually, hardware is an assemblage of components that operate continuously or discretely in time and interact via synchronous or asynchronous communication. Software is an assemblage of components that trade off use of a CPU, operating sequentially, and communicating by leaving traces of their (past and completed) execution on a stack or in memory.

A primary abstraction mechanism in software is the procedure (or the method). Procedures are terminating computations. The primary abstraction mechanism in hardware is a module, such as a chip, that operates fully concurrently with

the other components with which it is combined. These are very different abstraction mechanisms. Hardware modules do 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. Active objects are more like an afterthought, requiring still a model of computation to have any useful semantics. Hardware is active, more like processes than objects, but with a clear and clean semantics that is firmly rooted in the physical world.

Indeed, the synchronous abstraction that is widely used in hardware to build large, complex, and modular designs has more recently been applied to software [10]. Also, hardware models are conventionally constructed using hardware description languages such as Verilog and VHDL; these language realize a discrete-event model of computation that makes time a first-class concept, information shared by all components. Discrete-event models are sometimes used for software systems, particularly in the context of networking.

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 without 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.

A sophisticated component technology for embedded software will talk more about processes than procedures. It will talk about concurrency and the models of computation used to regulate interaction between components. And it will talk about time.

1.9 Problems not addressed

There are a number of interesting research problems related to embedded systems that we do not address here. Human-computer interaction, for example, is a key part of 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 models of computation. Finally, hardware and software design techniques that minimize power consumption are critical for portable devices. But I focus here on embedded software construction, leaving for someone else to set the agenda on these other important problems.

2.0 MODELS OF COMPUTATION

A model of computation is the "laws of physics" of concurrent components, including what they are (the ontology) how they communicate and how their flows of control are related (the protocols), and what information they share (the epistemology). It is their concurrent semantics.

Agha describes *actors*, which extend the concept of objects to concurrent computation [5]. 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. 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.

2.1 Concurrency

Design of networked embedded systems will require specification and modeling techniques 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, in sequential computation, the Von Neuman model of computation is a wildly successful universal abstraction. A key part of this success is that time is reduced 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. In practice, events are partially ordered at best. This makes it difficult to maintain a global notion of "system state," an essential part of the Von Neumann model.

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 [21], guarded communication [40], rendezvous, synchronous message passing, active messages [84], asynchronous message passing, streams (as in Kahn process networks [45]), dataflow (commonly used in signal and image processing), synchronous/reactive systems [10], Linda [18], and many others.

These abstractions partially or completely define a model of computation, the modular organizational and operational principles of a system. 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 [40]. Esterel [13], Lustre [33], Signal [11], and Argos [65] 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. 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, a networked 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.

The interaction between modules could follow any of several principles, e.g., those of Kahn process networks [45]. 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 [26] 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.

2.2 Examples of models of computation

There are a rich variety of models of computation that deal with concurrency and time in different ways. In this section, we outline some of the most useful models for embedded systems. All of these will lend a semantics to the same bubble-and-arc, or block-and-arrow diagram shown in figure 1. The bubbles are components and the arcs between them are their interconnections. The diagram suggests an epistemology where *A* knows about *C* but not about *B*.

2.2.1 Differential equations

One possible semantics for the syntax in figure 1 is that of differential equations. The arcs represent continuous functions of a continuum (time). The bubbles represent relations between these functions. 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 analog circuits and many physical systems. As such, they are certain to play a role in embedded systems, where sensors and actuators interact with the physical world. Embedded systems frequently contain components that are best modeled using differential equations, such as micro electromechanical systems, aeronautical systems, mechanical components, analog circuits, and microwave circuits. These components, however, interact with an electronic system that may serve as a controller and a recipient of sensor data. This electronic system may be digital, in which case there is a fundamental mismatch in models of computation. Joint modeling of a continuous subsystem with digital electronics is known as *mixed signal modeling*.

Differential equations form the model of computation used in Simulink, Saber, and VHDL-AMS, and are closely related to that in Spice circuit simulators.

2.2.2 Difference equations

Differential equations can be discretized to get difference equations, a commonly used model of computation in digital signal processing. This model of computation can be further generalized to support multirate difference equations. In either case, a global *clock* defines the discrete points at which signals have values (at the *ticks*).

Difference equations are considerably easier to implement in software than differential equations. Their key weaknesses are the global synchronization implied by the clock, and the awkwardness of specifying irregularly timed events and control logic.

2.2.3 Finite-state machines

In FSMs, bubbles represent system *state* and arcs represent state *transitions*. The simple FSM model of computation is not concurrent. Execution is a strictly ordered sequence of state transitions. Transition systems are a more general version, in that a given bubble may represent more than one system state (and there may be an infinite number of bubbles).

FSM models are excellent for 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

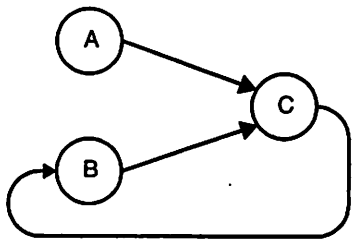

Figure 1. A single syntax (bubble-and-arc or block-and-arrow diagram) can have a number of possible semantics.

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 partially 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 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 a loose version of synchronous/reactive modeling (described below) with FSMs [34]. Statecharts have been adopted by UML for modeling the dynamics of software [24]. FSMs have also been combined with differential equations, yielding the so-called *hybrid systems* model of computation [39].

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 [29]. Thus, they present a promising model that is capable of abstracting program dynamics.

2.2.4 Synchronous/reactive models

In the synchronous/reactive (SR) model of computation [10], the arcs represent data values that are aligned with global clock ticks. Thus, they are discrete signals, as with difference equations, but unlike difference equations, a signal need not have a value at every clock tick. The bubbles represent relations between input and output values at each tick, and are usually partial functions with certain technical restrictions to ensure determinacy. Examples of languages that use the SR model of computation include Esterel [13], Signal [11], Lustre [19], and Argos [65]. Argos is a cleaner version of Statecharts that assumes SR concurrency semantics.

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.

2.2.5 Discrete-event models

In discrete-event (DE) models of computation, the arcs represent sets of *events* placed in time. An event consists of a *value* and *time stamp*. This model of computation is popular for specifying hardware and simulating telecommunications systems, and has been realized in a large number of simulation environments, simulation languages, and hardware description languages, including VHDL and Verilog. Unlike the SR model, there is no global clock tick, but like SR, differential equations, and difference equations, there is a globally consistent notion of time.

DE models are excellent descriptions of concurrent hardware, although increasingly the globally consistent notion of time is problematic. In particular, it over-specifies (or over-models) 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.

2.2.6 Cycle-driven models

Some systems with timed events are driven primarily by clocks, signals with events that are repeated indefinitely with a fixed time interval. Although discrete-event modeling for such systems is possible, it is costly, primarily due to the priority queue that sorts events chronologically. Cycle driven models associate components with clocks and stimulate computations regularly according to the clock ticks. This can lead to considerably more efficient execution.

In the Scenic system [59], for example, the 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 for system-level hardware design (see http://systemc.org).

2.2.7 Synchronous message passing

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) [40] and Milner's *calculus of communicating systems* (CCS) [68]. 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.

2.2.8 Asynchronous message passing

In asynchronous message passing, processes communicate by sending messages through channels that can buffer the messages. 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 those that

ensure determinate computation, namely Kahn process networks [45] and dataflow models.

In a process network (PN) model of computation, the arcs represent sequences of data values (tokens), and the bubbles 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, popular in signal processing, are a special case of process networks [57]. Dataflow models are also closely related to functional programming, although stylistically the two often look very different.

PN models are excellent for signal processing. 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 control logic.

Several special cases of PN are useful in certain circumstances. Dataflow models construct processes of a process network as sequences of atomic actor *firings*. Synchronous dataflow (SDF) is a particularly restricted special case with the extremely useful property that deadlock and boundedness are decidable [47][52][54][55]. Boolean dataflow (BDF) is a generalization that sometimes yields to deadlock and boundedness analysis, although fundamentally these questions remain undecidable [16]. Dynamic dataflow (DDF) uses only run-time analysis, and thus makes no attempt to statically answer questions about deadlock and boundedness [42][46][71].

2.2.9 Timed CSP and timed PN

CSP 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 [74]). In the Pamela system [82], 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.

2.2.10 Publish and subscribe

The publish and subscribe (PS) model of computation uses notification of events as the primary means of interaction between components. A component declares an interest in a family of events (subscribes), and another component asserts events (publishes). Some of the more sophisticated realizations of this principle are based on Linda [6][18], for example JavaSpaces from Sun Microsystems, which is built on top of a network-based distributed component technology called Jini [85]. A more elementary version can be found in the CORBA Event Service API.

The PS model of computation is well-suited to highly irregular, untimed communications. By "irregular" we mean both in time (sporadic) and in space (the publisher need not know who the subscribers are, and they can be constantly changing). Schmidt has extended the CORBA Event Service with notions of time (at the level of standard real-time operating systems), making the model particularly well suited for coordinating loosely coupled embedded components [79].

2.2.11 Unstructured events

The Java Beans, COM, and CORBA frameworks all provide a very loose model of computation that is based on method calls with no particular control on the order in which method calls occur. This is a highly flexible model of computation, and forms a good foundation for more restricted models of computation (such as the CORBA Event Service). It has a key advantage that since no synchronization is built in, then unsynchronized interactions can be easily implemented with no risk of deadlock. A major disadvantage, however, is that if synchronization is required, for example to enforce data precedences, then the programmer must build up the mechanisms from scratch, and maintaining determinacy and avoiding deadlock become difficult.

2.3 Choosing models of computation

The rich variety of concurrent 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.

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 [58].

Many researchers have thought deeply about the role of time in computation. Benveniste *et al.* 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" [11]. Lamport observes that a coordinated notion of time cannot be exactly maintained in distributed systems, and shows that a partial ordering is sufficient [50]. 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 [25]. 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" [67]. 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, but such a mixture would be extremely complex and difficult to use, and synthesis and validation tools would be difficult to design.

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 allocation. Finite-state machines are good at modeling at least simple control logic, but inadequate for modeling data dependencies in 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 [8] and Rapide [62], 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 intend take the stand that asynchronous message passing is a bad idea for the designs it addresses!

2.4 Visual syntaxes and visualizing dynamics

A major part of designing robust networked embedded systems will be ensuring that the designers understand their behavior. Achieving this understanding in the face of mutating software architecture, concurrency, and real time will be difficult. Visual depictions of systems have always held a strong human appeal, making them extremely effective in conveying information about a design. Many of the models of computation described above can use such depictions to completely and formally specify models.

These visual depictions offer an alternative *syntax* to associate with the semantics of a model of computation. 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. Schematics are often replaced today by text in hardware description languages such as VHDL or Verilog. In other contexts, visual representations have largely failed, for example flowcharts for capturing the behavior of software. Recently, a number of innovative visual formalisms have been garnering support, including visual dataflow, hierarchical concurrent finite state machines, and object models. The UML visual language for object modeling, for example, has been receiving a great deal of attention [24].

A subset of visual languages that are recognizable as "block diagrams" represent concurrent systems in domain-specific ways. There are many possible concurrency semantics (and many possible models of computation) associated with such diagrams. Formalizing the semantics is essential if these diagrams are to be used for embedded system specification and design.

3.0 REAL TIME

Virtually all 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.

Some such embedded systems incorporate a real-time operating system, which in addition to standard operating system services such as I/O, offers specialized scheduling services tuned to real-time needs. The schedules might be based on priorities, using for example the principles of rate-monotonic scheduling [60][48], 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.

3.1 Reactive Systems

Reactive systems are those that react continuously to their environment at the speed of the environment. Harel and Pnueli [36] 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 have emerged to provide more robust support for reactive system design than what is provided by real-time operating systems. The synchronous languages, such as Esterel [13], Lustre [33], Signal [11], and Argos [65], 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, and Esterel is used by Dasa. 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.

Most of these uses of synchronous languages have only needed fairly small-scale monolithic programs, although there have been investigations into distributed versions. For example, extensions to Esterel support processes that do not obey the synchronous hypothesis [14]. Processes communicate via channels with rendezvous. This has been subsequently extended to add process suspension and resumption.

These 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.

3.2 Scheduling

Although scheduling is an old topic, it is certainly not played out. A real-time scheduler provides some assurances given certain properties of the components, such as the period of their periodic invocation, or the deadlines associated with certain tasks. Rate monotonic scheduling principles [60][48] translate the first of these into priorities. Priorities may also be given based on semantic information about the application, for example reflecting the criticality with which some event must be dealt with.

A key problem in scheduling is that most methods are not compositional. That is, even if assurances can be provided individually to a pair of components, there are no systematic mechanisms for providing assurances to the aggregate of the two, 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 entering 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 is effectively blocked by the medium 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) 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.

4.0 COMPONENT INTERFACES

A proper agenda for research should detail some specific promising directions, some for the near term and some more speculative. On the more speculative side, I believe that type system concepts can be extended to drastically change the design of concurrent components in a real-time context. 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 or by lossless convertibility. 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 [80]. I suggest first that dynamic properties of an interface, such as the protocols used by a component to interact with other components, can be described using nondeterministic automata, and that the pertinent partial ordering relation is the simulation relation between automata. I also speculate that various timed automata extensions can perhaps be used in similar ways to define much more completely the temporal properties of an interface than what is common practice today.

4.1 Strongly typed languages

Type systems in modern languages serve to promote safety through static (compile time) and dynamic (run time) checking. In a computation environment, two kinds of run-time errors can occur, trapped errors and untrapped errors. Trapped errors cause the computation to stop immediately. A run-time handler can attempt to recover gracefully. Untrapped errors, which may go unnoticed (for a while) and

later cause arbitrary behavior, can be disastrous for an embedded system. Moreover, they are less likely to be detected in testing. Examples of untrapped errors in many general purpose languages are jumping to the wrong address, or accessing data past the end of an array.

Strongly typed languages help prevent both trapped and untrapped errors. Many errors are detected at compile time, and run-time support for the type system can help ensure that the remaining errors are trapped. This helps prevent arbitrary behavior, but it only deals with certain aspects of program behavior. Moreover, run-time support for the type system, which can be provided systematically through preconditions and contracts, may incur substantial overhead.

Modern languages, such as Java and ML, emphasize avoiding untrapped errors. There is significant run-time overhead incurred in the required safety checks. Several researchers have shown that in many cases, this overhead can be eliminated through compile-time analysis (see for example [87]). The approach is to augment the type system to include such properties as array size, and then to annotate the generated code with assertions of safety. A run-time environment can thus bypass the safety checks.

Ousterhout [70] argues that strong typing compromises modularity and discourages reuse.

> "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."

The alternative he advocates is languages without strong typing, such as Lisp and Tcl, where safety can only be achieved by extensive run-time checking. However, since type checking is postponed to the last possible moment, the system does not have fail-stop behavior, so a system may exhibit erroneous behavior only after running for an extended period of time after the error has occurred. Identifying the source of the problem can be difficult, and guaranteeing the code may be impossible.

Ousterhout raises a valid point, but the solution is not to discard strong typing. Particularly for embedded systems, the extra degree of safety offered by strong typing overwhelms even the desire for modularity and reuse. How can we achieve modularity and reuse without discarding strong typing? One solution is to use polymorphism, reflection, and run-time type inference and type checking.

Strong typing and type resolution have other benefits in addition to the ones mentioned above. Strong typing helps to clarify the interfaces of components and makes libraries more manageable. Just as typing may improve run-time efficiency in a general-purpose language by allowing the compiler to generate specialized code, type information can be used for efficient synthesis of embedded hardware and software configurations. For example, if the type checker asserts that a certain polymorphic component will only receive integers, and that component is to be implemented in configurable hardware, then only hardware dealing with integers needs to be synthesized.

In general-purpose strongly-typed languages, such as C++ and Java, static type checking done by the compiler can find a large fraction of program errors in object-oriented programs. However, with networked embedded systems where parallel execution, agents, migrating code, and software upgrades are all possibilities, static type checking does not do enough. Some of the type checking must be done at run time. Java's run-time type identification (RTTI) system together with its reflection package specifically addresses this problem by supporting run-time queries of type constraints and run-time verification of type compatibility.

Type systems in modern programming languages, however, do not go far enough. Many errors that in principle may be detectable at compile time are not within the scope of the type system. Several researchers have proposed extending the type system to handle such errors as array bounds overruns, which are traditionally left to the run-time system [87]. But many are still not dealt with. For example, the communication protocols between concurrent processes are not type checked. Yet failures in concurrency and synchronization are common causes of critical system failures in embedded systems.

4.2 Process-level type systems

The first-order mechanism for assurance is that the designer understand the system. But generally the designer needs a great deal of help. Object-oriented programming, for example, helps a designer understand the static structure of a software architecture by providing syntactic features of the language supporting object-oriented design, and by providing a compiler that checks types. I suggest that extended types that include dynamic properties of an interface will require novel syntactic language support as well as new compiler and run-time techniques.

The best parts of UML (especially the static structure diagrams) are those that are directly supported by the language syntaxes used by designers (especially C++ and Java). The syntactic structure of a program directly reflects the object model, and the compiler assures consistency in the model. The weakest parts of UML (especially its variant of Statecharts for state diagrams and its modeling of concurrency) are those with no syntactic support in the widely used languages. Few tools are available for ensuring consistency between programs and their models and for validating the models. Perhaps eventually code generation from these models will ameliorate this, although this really amounts to defining new languages with graphical syntaxes, a non-trivial challenge.

Extended type systems could, in principle, capture the following aspects of a system:

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

This can be done without modifying the underlying languages, but rather by overlaying on standard languages design patterns that make these types explicit. These will be polymorphic, offering a systematic approach to tolerant interfaces. A key part of the research here is to identify the

syntactic language support for such types.

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 [61]. Martin-Löf introduces *dependent types*, in which types are indexed by terms [66]. 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 [87]. The technique uses singleton types instead of general terms [38] to help avoid undecidability. While much of the fundamental work has been developed using functional languages (especially ML), there is no reason that I can see that it cannot be applied to more widely accepted languages.

Another innovative use of type systems is that of Necula, who describes the use of proof-carrying code [69]. Here, a program includes with it a proof of validity or compliance to some requirement, such as safety. If the code type checks, then it is valid. This is used primarily for security. The main drawback appears to be in the difficulty of constructing the proofs. We may face a similar drawback in our use of dependent types for capturing real-time properties in that constructing the real-time properties may prove difficult.

4.3 On-line type system

Static support for type systems give the compiler responsibility for the robustness of software [17]. This is not adequate when the software architecture is dynamic. The software needs to take responsibility for its own robustness [49]. 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" [31][81][86]. It was the first language to use type inference in an integrated way [41], 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 [78]. 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 inequalities [68]. Reasonably efficient algorithms have been identified for solving these systems of inequalities [75], 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 a notable one of these [13]. Esterel compilers synthesize run-time algorithms that converge to a fixed point at each clock of a synchronous system [11]. Such synthesis requires detailed static information about the structure of the application, but methods have been demonstrated that use less static information [23]. Although these techniques have not been proposed primarily in the context of a type system, I believe they can be adapted.

4.4 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 program state. Thus, we need reflection on the program dynamics.

The first question becomes at what granularity to do this. Reflection intrinsically refers to a particular abstracted representation of a program. E.g., in the case of static structure, Java's reflection package does not include finer granularity than methods, nor coarser granularity than objects.

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 communication 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 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 [29]. Some variations of Statecharts support concurrency using models that are different from those in the original Statecharts [65][83]. 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. For example, a component that derives from another inherits its automaton and refines the states of the automaton (similar to the hierarchy, or "or" states in Statecharts).

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 to reflect aspects of the application that transcend a single component or aggregate of closely related components.

5.0 FRAMEWORK FRAMEWORKS

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.

There are several ways to mix frameworks. One is through specialization (analogous to subtyping) where one framework is simply a more restricted version of another. For example, a Unix application that involves multiple processes might use pipes in situations where the benefits of pipes are desired. But it might not always use pipes, or not use pipes throughout the application. The subsystem that uses pipes, ideally, is assured the benefits of pipes, such as freedom from deadlock. The rest of the system has no such assurance.

A second way to mix frameworks is hierarchically. A component in one framework is actually an aggregate of components in another. This is the approach taken in the Ptolemy project [20]. The challenge here is to avoid having to design each pairwise hierarchical combination of frameworks.

The approach we take in the Ptolemy project is to use a system-level type concept that we call *domain polymorphism*. In Ptolemy software, a model of computation is realized by a software infrastructure called a domain. A component that is domain polymorphic is one that can operate in a number of domains. The objective is that the interface exposed by an aggregate of components in a domain is itself domain polymorphic, and thus the aggregate can be used in any of several other domains with clear semantics.

Initially, we constructed domain polymorphic components in an ad hoc fashion, using intuition to define an interface that was as unspecific as possible. More recently we have been characterizing these interfaces using nondeterministic automata to given precisely the assumptions and requirements of the interface. The services provided by each domain are also characterized by automata. A component can operate within a domain if its interface automata simulate those of the domain.

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 framework, one that mixes modeling techniques heterogeneously [2]. A model in a domain is called a facet, and heterogeneous models are multi-facetted designs [7]. Jourdan *et al.* have proposed a combination of Argos, a hierarchical finite-state machine language, with Lustre [33], which has a more dataflow flavor, albeit still within a synchronous/reac-

tive concurrency framework [44]. Another interesting integration of diverse semantic models is done in Statemate [35], 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.

6.0 ACKNOWLEDGEMENTS

The contents of this paper are strongly influenced by the St. John *Workshop on Software Development for the Post-PC World*, December, 1999, and the St. Thomas *Workshop on Software Behavior Description*, December, 1998. Particular thanks to Cordell Green, Tom Henzinger, Paul Hudak, Gregor Kiczales, Bob Laddaga, Bill Mark, John Mitchell, Bill Scherlis, Victoria Stavridou and Janos Sztipanovits, and to Bultler Lampson for asking the right hard questions. Many of the technical ideas are drawn from the Ptolemy Project (http://ptolemy.eecs.berkeley.edu), and have been particularly influenced by Shuvra Bhattacharyya, Joe Buck, John Davis, Steven Edwards, Soonhoi Ha, Christopher Hylands, Bart Kienhuis, Bilung Lee, Jie Liu, Praveen Murthy, Steve Neuendorffer, John Reekie, Kees Vissers, and Yuhong Xiong.

REFERENCES

[1] S. Abramsky, S. J. Gay and R. Nagarajan. "Interaction Categories and the Foundations of Typed Concurrent Programming." In: *Deductive Program Design: Proceedings of the 1994 Marktoberdorf Summer School* (M. Broy, ed.). NATO ASI Series F, Springer-Verlag, 1995.

[2] N. Abu-Ghazaleh, P. Alexander, D. Dieckman, R. Murali, and J. Penix, "Orbit — A Framework for High Assurance System Design and Analysis," University of Cincinnati, TR 211/01/98/ECECS, 1998.

[3] G. A. Agha, "Concurrent Object-Oriented Programming," *Communications of the ACM*, 33(9), pp. 125-141, 1990.

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

[5] G. A. Agha, "Abstracting Interaction Patterns: A Programming Paradigm for Open Distributed Systems," in *Formal Methods for Open Object-based Distributed Systems*, IFIP Transactions, E. Najm and J.-B. Stefani, Eds., Chapman & Hall, 1997.

[6] S. Ahuja, N. Carreiro, and D. Gelernter, "Linda and Friends," *Computer*, Vol. 19, No. 8, Aug. 1986, pp. 26-34.

[7] P. Alexander, "Multi-Facetted Design: The Key to Systems Engineering," in *Proceedings of Forum on Design Languages (FDL-98)*, September, 1998.

[8] R. Allen and D. Garlan, "Formalizing Architectural Connection," in *Proc. of the 16th International Conference on Software Engineering* (ICSE 94), May 1994, pp. 71-80, IEEE Computer Society Press.

[9] R. Ben-Natan, *CORBA: A Guide to Common Object Request Broker Architecture*, McGraw-Hill, Inc., ISBN 0-07-005427-4, 1995.

[10] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems," *Proceedings of the IEEE*, vol. 79, no. 9, 1991, pp. 1270-1282.

[11] A. Benveniste and P. Le Guernic, "Hybrid Dynamical Systems Theory and the SIGNAL Language," *IEEE Trans. on Automatic Control*, vol. 35, no. 5, pp. 525-546, May 1990.

[12] G. Berry, "Real Time programming: Special purpose or general purpose languages," in *Information Processing*, Ed. G. Ritter, Elsevier Science Publishers B.V. (North Holland), vol. 89, pp. 11-17, 1989.

[13] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87-152, 1992.

[14] G. Berry, A. Ramesh, R. K. Shyamasundar, "Communicating Reactive Processes," *20th ACM Symp. on Principles of Programming Languages*, Charleston, January 1993.

[15] S. S. Bhattacharyya, P. K. Murthy and E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, Norwell, Mass, 1996.

[16] J. T. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*, Tech. Report UCB/ERL 93/69, Ph.D. Dissertation, Dept. of EECS, University of California, Berkeley, CA 94720, 1993.

[17] L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computing Surveys*, Vol. 17, No. 4, pp. 471-522, 1985.

[18] N. Carriero and D. Gelernter, "Linda in Context," *Comm. of the ACM*, vol. 32, no. 4, pp. 444-458, April 1989.

[19] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "LUSTRE: A Declarative Language for Programming Synchronous Systems," *Conference Record of the 14th Annual ACM Symp. on Principles of Programming Languages*, Munich, Germany, January, 1987.

[20] J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay and Y. Xiong, "Overview of the Ptolemy Project," ERL Technical Report UCB/ERL No. M99/37, Dept. EECS, University of California, Berkeley, CA 94720, July 1999.

[21] E. Dijkstra, "Cooperating Sequential Processes", in *Programming Languages*, E F. Genuys, editor, Academic Press, New York, 1968.

[22] B. P. Douglass, *Real-Time UML*, Addison Wesley, 1998

[23] S. A. Edwards, "The Specification and Execution of Heterogeneous Synchronous Reactive Systems," Ph.D. thesis, University of California, Berkeley, May 1997. Available as UCB/ERL M97/31. (http://ptolemy.eecs.berkeley.edu/papers/97/sedwardsThesis/)

[24] H.-E. Eriksson and M. Penker, *UML Toolkit*, Wiley, 1998.

[25] C. J. Fidge, "Logical Time in Distributed Systems," *Computer*, Vol. 24, No. 8, pp. 28-33, Aug. 1991.

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

[27] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1994.

[28] A. J. C. van Gemund, *"Performance Prediction of Parallel Processing Systems: The PAMELA Methodology,"* Proc. 7th Int. Conf. on Supercomputing, pages 418-327, Tokyo, July 1993.

[29] A. Girault, B. Lee, and E. A. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models," *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*, Vol. 18, No. 6, June 1999.

[30] Goossens, G., Lanneer, D., Pauwels, M., Depuydt, F., Schoofs, K. Kifli, A., Cornero, M., Petroni, P., Catthoor, F., and de Man, H., "Integration of medium-throughput signal processing algorithms on flexible instruction-set architectures," *Journal of VLSI Signal Processing*, Jan. 1995, vol. 9, No. 1-2, p. 49-65.

[31] M. J. Gordon, R. Milner, L. Morris, M. Newey and C. P. Wadsworth, "A Metalanguage for Interactive Proof in LCF," *Conf.Record of the 5th Annual ACM Symp. on Principles of Programming Languages*, ACM, pp. 119-130, 1978.

[32] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, Dordrecht, 1993.

[33] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE," *Proc. of the IEEE*, Vol. 79, No. 9, 1991, pp. 1305-1319.

[34] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Sci. Comput. Program.*, vol 8, pp. 231-274, 1987.

[35] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, M. Trakhtenbrot, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Trans. on Software Engineering*, Vol. 16, No. 4, April 1990.

[36] D. Harel and A. Pnueli, "On the Development of Reactive Systems," in *Logic and Models for Verification and Specification of Concurrent Systems*, Springer Verlag, 1985.

[37] R. Harper and P. Lee, "Advanced Languages for Systems Software: The Fox Project in 1994," Technical Report, CMU-CS-FOX-94-01, Carnegie-Mellon University, 1994.

[38] S. Hayashi, "Singleton, Union, and Intersection Types for Program Extraction," in A. R. Meyer (ed.), *Proc. of the Int. Conf. on Theoretical Aspects of Computer Science*, pp. 701-730, 1991.

[39] T. A. Henzinger, "The theory of hybrid automata," in *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, IEEE Computer Society Press, 1996, pp. 278-292, invited tutorial.

[40] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8, August 1978.

[41] P. Hudak, "Conception, Evolution, and Application of Functional Programming Languages," *ACM Computing Surveys*, Vol. 21, No. 3, September 1989.

[42] R. Jagannathan, "Parallel Execution of GLU Programs," presented at *2nd International Workshop on Dataflow Computing*, Hamilton Island, Queensland, Australia, May 1992.

[43] R. E. Johnson, "Frameworks = (Components + Patterns)," *Communications of the ACM*, Vol. 40, No. 10, pp. 39-42, October 1997.

[44] M. Jourdan, F. Lagnier, F. Maraninchi, and P. Raymond, "A Multiparadigm Language for Reactive Systems," in *Proc. of the 1994 Int. Conf. on Computer Languages*, Toulouse, France, May 1994.

[45] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," Proc. of the IFIP Congress 74, North-Holland Publishing Co., 1974.

[46] D. J. Kaplan, *et al.*, "Processing Graph Method Specification Version 1.0," Unpublished Memorandum, The Naval Research Laboratory, Washington D.C., December 11, 1987.

[47] R. M. Karp, R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM Journal*, Vol. 14, pp. 1390-1411, November, 1966.

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

[49] R. Laddaga, "Active Software," position paper for the *St. Thomas Workshop on Software Behavior Description*, December, 1998.

[50] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, Vol. 21, No. 7, July, 1978.

[51] P. Lapsley, J. Bier, A. Shoham, and E. A. Lee, *DSP Processor Fundamentals — Architectures and Features*, IEEE Press, New York, 1997.

[52] R. Lauwereins, P. Wauters, M. Adé, J. A. Peperstraete, "Geometric Parallelism and Cyclo-Static Dataflow in GRAPE-II", *Proc. 5th Int. Workshop on Rapid System Prototyping*, Grenoble, France, June, 1994.

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

[54] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, September, 1987.

[55] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Trans. on Computers*, January, 1987.

[56] E. A. Lee and D. G. Messerschmitt, *Digital Communication*, Second Edition, Kluwer Academic Press, Norwood, Mass, 1994.

[57] E. A. Lee and T. M. Parks, "Dataflow Process Networks,", *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773-801, May, 1995.

[58] E. A. Lee and A. Sangiovanni-Vincentelli, "A Framework for Comparing Models of Computation," *IEEE Transaction on CAD*, December 1998.

[59] S. Liao, S. Tjiang, R. Gupta, "An efficient implementation of reactivity for modeling hardware in the Scenic design envi-

ronment," *Proc. of the Design Automation Conference (DAC 97)*, Anaheim, CA, 1997.

[60] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *JACM*, vol. 20, pp. 46--61, January 1973.

[61] J. M. Lucassen and D. K. Gifford, "Polymorphic Effect Systems," in *Proc. 15-th ACM Symp. on Principles of Programming Languages*, pp. 47-57, 1988.

[62] D. C. Luckham and J. Vera, "An Event-Based Architecture Definition Language," *IEEE Transactions on Software Engineering*, 21(9), pp. 717-734, September, 1995.

[63] N. A. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

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

[65] F. Maraninchi, "The Argos Language: Graphical Representation of Automata and Description of Reactive Systems," in *Proc. IEEE Workshop on Visual Languages*, Kobe, Japan, Oct. 1991.

[66] P. Martin-Löf, "Constructive Mathematics and Computer Programming," in *Logic, Methodology, and Philosophy of Science VI*, pp. 153-175, North-Holland, 1980.

[67] F. Mattern, "Virtual Time and Global States of Distributed Systems," in *Parallel and Distributed Algorithms*, M. Cosnard and P. Quinton, eds., North-Holland, Amsterdam, 1989, pp. 215-226.

[68] R. Milner, *A Theory of Type Polymorphism in Programming*, Journal of Computer and System Sciences 17, pp. 384-375, 1978.

[69] G. Necula, "Proof-Carrying Code," in *Conf. Record of the 24th Annual ACM Symp. on Principles of Programming Languages*, pp. 106-119, ACM Press, 1997.

[70] J. K. Ousterhout, "Scripting: Higher Level Programming for the 21 Century," *IEEE Computer*, March 1998.

[71] T. M. Parks, *Bounded Scheduling of Process Networks*, Technical Report UCB/ERL-95-105. PhD Dissertation. EECS Department, University of California. Berkeley, CA 94720, December 1995.

[72] J. Peterson and K. Hammond (eds), *Haskell 1.3: A Non-strict, Purely Functional Language*, Yale University Research Report YALEU / DCS / RR-1106, May 1996.

[73] V. R. Pratt, "Modeling Concurrency with Partial Orders," *Int. Journal. of Parallel Programming*, vol. 15, no. 1, pp. 33-71, Feb. 1986.

[74] G. M. Reed and A. W. Roscoe, "A Timed Model for Communicating Sequential Processes," *Theoretical Computer Science*, 58(1/3): 249-261, June 1988.

[75] J. Rehof and T. Mogensen, "Tractable Constraints in Finite Semilattices," *Third International Static Analysis Symposium*, pp. 285-301, Volume 1145 of Lecture Notes in Computer Science, Springer, Sept., 1996.

[76] W. Reisig, *Petri Nets: An Introduction*, Springer-Verlag, 1985.

[77] M. Schlett, "Trends in Embedded-Microprocessor Design," *Computer*, vol. 31, No. 8, pp. 44-49, August 1998.

[78] D. Scott, "Outline of a mathematical theory of computation", *Proc. of the 4th annual Princeton conf. on Information sciences and systems*, 1970, 169-176.

[79] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design of the TAO Real-Time Object Request Broker," *Computer Communications*, Elsivier Science, Volume 21, No 4, April, 1998.

[80] W. T. Trotter, *Combinatorics and Partially Ordered Sets*, Johns Hopkins University Press, Baltimore, Maryland, 1992.

[81] J. D. Ullman, *Elements of ML Programming*, Prentice-Hall, 1994.

[82] A. J. C. van Gemund, *"Performance Prediction of Parallel Processing Systems: The PAMELA Methodology,"* Proc. 7th Int. Conf. on Supercomputing, pages 418-327, Tokyo, July 1993.

[83] M. von der Beeck, "A Comparison of Statecharts Variants," in *Proc. of Formal Techniques in Real Time and Fault Tolerant Systems, LNCS* 863, pp. 128-148, Sprinter-Verlag, Berlin, 1994.

[84] T. von Eicken, D. E. Culler, and S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communications and computation," *Proc. of the 19th Int. Symp. on Computer Architecture*, Gold Coast, Australia, May 1992, also available as technical report TR UCB/CSD 92/675, CS Division, University of California, Berkeley, CA 94720.

[85] J. Waldo, "Jini™ Architecture Overview," Sun Microsystems, http://www.sun.com/products/jini, 1998.

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

[87] H. Xi and F. Pfenning, "Eliminating Array Bound Checking Through Dependent Types," In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98)*, pp. 249-257, Montreal, June 1998.