

Computing for Embedded Systems

Edward A. Lee

UC Berkeley, Berkeley, CA 94720, USA, eal@eecs.berkeley.edu

Abstract – *Embedded software is increasingly a composition of concurrent components. Components in such systems interact in a rich variety of ways, not limited to the simple transfer of control of method calls in object-oriented design. I describe a view where the systems are modeled as assemblages of components within one of several models of computation, where components with well-defined interfaces are composed. The declaration of these component interfaces becomes a central problem, and the composition of properties becomes the central benefit. Unlike object-oriented interfaces, these interfaces must declare dynamic properties such as communication protocols and temporal properties. The model of computation must include shared information, such as time (a total ordering constraint), or causality (a partial ordering constraint). The properties of a model of computation strongly determine the problems that it matches, and frequently, practical systems are forced to use multiple models of computation. I briefly describe how this can be done systematically.*

Keywords – *Embedded software, models of computation, concurrency, real time, frameworks, architectural patterns.*

I. MODELS OF COMPUTATION

What is a model of computation? I will give you an ostensive definition of Wittgenstein, not an essential definition of Socrates. To give you an essential definition, I would have to identify one or more properties shared by everything that is a model of computation, and only by things that are models of computation. Wittgenstein legitimized the notion that for many perfectly understandable linguistic constructs, such definitions are not possible. An ostensive definition is one constructed by examples. It reflects the way we learn the meaning of a great many words, such as “beauty,” “integrity.” “Model of computation” is not a Platonic form. It is instead an element of a language game. Understanding of the concept is demonstrated by being able to use the phrase, not by being able to define it.

A model of computation governs the interaction of components in a design. In the classical Von Neumann model of computation, the components are statements that transform the state of a data store. Their interaction is via the store. An early elaboration of this model is found in the use of procedures to define components with coarser granularity than statements. A later elaboration is found in the use of objects to divide the store into pieces and to bundle those pieces with the procedures that operate on them to get objects.

In the functional model of computation, the components are functions, and their interaction is through function

composition. There are many variants of this model computation.

Embedded software design differs from these two classical models in its handling of concurrency. Components are concurrent processes or threads, typically scheduled by a priority-driven real-time operating system. (The difference between a “process” and a “thread” is that threads share a common data store, whereas processes do not.) In either case, the processes or threads communicate, and communication is regulated through the use of semaphores and mutual exclusion mechanisms (mutexes). Processes may also be executing on distinct processors.

The aim of this paper is to characterize the current practice of embedded software design as a model of computation, and to explore variants of this model of computation that may yield desirable properties for systems.

II. PATTERNS

Software engineering has advanced considerably in recent years through the use of object modeling (Booch *et. al*, 1999) and design patterns (Gamma, *et. al*, 1994). Although design patterns are often realized in idiomatic uses of conventional programming languages (Lea, 1997), they raise the level of abstraction above the programming language constructs.

Object modeling provides the lexicon for design patterns. UML (Booch, *et. al*, 1999), in particular, is a popular notation for object modeling, and is often used to define and describe design patterns. This lexicon has been used to describe the patterns of concurrent programming (Schmitt, *et. al*, 2000) and real-time programming (Douglass, 1998 and 1999).

Some authors distinguish between design patterns and architectural patterns (Buschmann, *et. al* 1996). An architectural pattern “expresses a fundamental structural organizational schema for software systems” (Schmidt, *et. al*, 2000). Models of computation are architectural patterns, focused particularly on the operational relationships between concurrent or sequential components.

Patterns serve as guidelines for programmers, and models of computation as the underlying principles. A third element is a *framework*, which enforces the patterns and implements the models of computation.

III. FRAMEWORKS

In this context, a framework is a set of constraints on components and their interaction, and a set of benefits that derive from those constraints. This is broader than, but consistent with the definition of frameworks in object-oriented design (Johnson, 1997). It is stronger than design patterns in that it imposes constraints, whereas patterns describe practice in a largely unconstrained context. It is similar to architectural patterns, but I wish to emphasize the constraining nature of the framework, and the ensuing benefits of those constraints. The term “architectural patterns” does not seem to me to suggest constraints, but rather convention.

By this definition, there are a huge number of frameworks in common use. 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 and DCOM are frameworks, although their constraints are weaker than those of operating systems or programming languages (it is easy, for example, in a system primarily based on CORBA, to circumvent CORBA and create a socket connection). SystemC, which supports synchronous digital hardware design, is a framework (see <http://systemc.org>). Synchronous digital hardware design itself is an architectural pattern. Synchrony itself, with a well-defined semantics, is a model of computation. Java Beans define an architectural pattern supported by a number of frameworks. A particular class library and policies for its use is a framework, although once again, such frameworks rarely have mechanisms for strongly enforcing their constraints.

In this paper, I aim to illustrate how frameworks can be built to support useful architectural patterns based on well-understood models of computation for embedded real-time system design.

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? In a given ontology, 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 is 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.

One 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.

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 an integral part of the framework semantics.

IV. ARCHITECTURE DESCRIPTION LANGUAGES

Certain architecture description languages (ADLs), such as Wright (Allen and Garlan, 1994) and Rapide (Luckham and Vera, 1995), define a model of computation. The models are intended for describing the rich sorts of component interactions that commonly arise in software architecture. Indeed, such descriptions often yield good insights about design. But sometimes, the match is poor. Wright, for example, which is based on CSP, does not cleanly describe asynchronous message passing (it requires giving detailed descriptions of the mechanisms of message passing).

I believe that what we really want are architecture design languages rather than architecture description languages. That is, their focus should not be on describing current practice, but rather on improving future practice. Wright, therefore, with its strong commitment to CSP, should not be concerned with whether it cleanly models asynchronous message passing. It should instead take the stand that asynchronous message passing is a bad idea for the designs it addresses.

V. PROGRAMMING LANGUAGES

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 (Hoare, 1978). Esterel (Berry and Gonthier, 1992), Lustre (Halbwachs, *et. al.*, 1991), Signal (Benveniste and Le Guernic, 1990), and Argos (Maranichi, 1991) support the synchronous/reactive model. These languages are based on a very elegant model of computation with a rich set of formal properties. Nonetheless, acceptance is slow, probably because the model of computation is unfamiliar to many programmers, 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 can follow any of several principles, e.g., those of Kahn process networks (Kahn 1974). This abstraction provides a robust interaction layer with loosely synchronized communication and support for mutable systems (in which subsystems come and go). These features are not directly built into any of the underlying languages, but rather interact with program components as an application interface (API). The programmer uses them as a design pattern rather than as a language feature. This makes acceptance somewhat easier, but the framework is weaker, so designers that have not completely bought into the model of computation will circumvent it and get unexpected results.

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. SystemC, for example, combines synchronous, cycle-driven execution with asynchronous, event-driven interaction between synchronous islands. Thus, domain-specific approaches can be combined. Rosetta (see <http://www.sldl.org/>), is perhaps the first programming language to combine specifications of components and their interactions with specification of the interaction semantics.

VI. EXAMPLES OF MODELS OF COMPUTATION

There are many models of computation that deal with concurrency and time. In this section, we outline some of the most useful ones for embedded systems. In all cases, the model of computation defines components and their interaction. In all cases, the model of computation has both strengths and weaknesses. It will be applicable in some situations, and awkward in others.

Continuous time and differential equations

One possible semantics for components and their interaction is that of differential equations. The components represent relations between continuous-time functions, and the interactions are the continuous-time functions themselves. 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.

Discrete time and 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.

State machines

In finite state machines (FSMs), components represent system state and the interactions 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 component may represent more than one system state (and there may be an infinite number of components). Abstract state machines (ASMs) elaborate the description of states in that components (states) represent algebraic systems, and the interactions represent transformations of these algebraic systems (Gurevich, 1997).

State machine 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 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 is ameliorated 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 (Harel, 1987). Statecharts have been adopted by UML for modeling the dynamics of software (Booch, *et. al*, 1999). FSMs have also been combined with differential equations, yielding the so-called hybrid systems model of computation (Henzinger, 1996).

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 (Girault, *et. al*, 1999). They present a

promising model that is capable of abstracting program dynamics.

Synchronous/reactive models

In the synchronous/reactive (SR) model of computation (Benveniste and Berry, 1991), the interactions between components are via data values that are aligned with global clock ticks. A sequence of such values, one for each tick, is a discrete signal, as with difference equations. But unlike discrete-time models, a signal need not have a value at every clock tick.

The components 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, Signal, Lustre, and Argos, all mentioned earlier.

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.

Discrete-event models

In discrete-event (DE) models of computation, the interactions between components are via events placed on a time line. A signal is a sequence of such events. The components process events in chronological order.

The DE model of computation is popular for specifying hardware and simulating telecommunications systems, and has been realized in a large number of frameworks, 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.

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 (Liao, *et. al*, 1997), 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>).

Giotto (Henzinger, *et. al*, 2000), is a new, experimental language for periodic, hard-real-time systems that are to be implemented on distributed, time-driven architectures such as the TTA (Kopetz, *et. al*, 2000).

Rate monotonic scheduling

A very commonly used model of computation in embedded systems design is rate-monotonic scheduling (Klein, *et. al*, 1993), first introduced by Liu and Leland. In RMS, tasks are assumed to have periodic actions, and they run as processes under the control of a priority-driven preemptive scheduler. This model is so popular that it is routinely applied even when tasks are not periodic in nature, and priorities are tweaked until the application seems to work.

A significant drawback in this model of computation is that it does not intrinsically include any interaction mechanisms between components, unlike, say, synchronous/reactive models, where variables written by one component can be read by another. Consequently, designers implement communication strategies of their own, typically using mutual exclusion to access shared data repositories. These mutexes, however, interact in undesirable ways with the preemptive scheduling, yielding such artifacts as priority inversion. Reasoning about the interactions between priorities and other interaction schemes such as mutual exclusion can be very difficult, and the result is often fragile designs.

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 uninterrupted step. Examples of rendezvous models include Hoare’s communicating sequential processes (CSP) (Hoare, 1978) and Milner’s calculus of communicating systems (CCS) (Milner, 1978). 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.

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 (Kahn, 1974) and dataflow models.

In a process network (PN) model of computation, the interactions between components are via sequences of data values (tokens), and the components represent functions that map input sequences into output sequences. Certain technical restrictions on these functions are necessary to ensure determinacy, meaning that the sequences are fully specified. Dataflow models, popular in signal processing, are a special case of process networks (Lee and Parks, 1995). 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 (Lee and Messerschmitt, 1987) Boolean dataflow (BDF) is a generalization that sometimes yields to deadlock and boundedness analysis, although fundamentally these questions remain undecidable (Buck, 1993). Dynamic dataflow (DDF) uses only run-time

analysis, and thus makes no attempt to statically answer questions about deadlock and boundedness (Parks, 1995).

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 Reed and Roscoe, 1988). In the Pamela system (van Gemund, 1993), 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.

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 (Carriero and Gelernter, 1989), for example JavaSpaces from Sun Microsystems, which is built on top of a network-based distributed component technology called Jini. 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 (Schmidt, *et. al.*, 1998).

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, 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.

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 (Lee and Sangiovanni-Vincentelli, 1998).

Many researchers have thought deeply about the role of time in computation. Benveniste and Le Guernic observe that in certain classes of systems, “the nature of time is by no means universal, but rather local to each subsystem, and consequently multiform” (1990). Lamport observes that a coordinated notion of time cannot be exactly maintained in distributed systems, and shows that a partial ordering is sufficient (Lamport, 1978). 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 (Fidge, 1991). A partial ordering relationship between these lists of times is determined by process creation, destruction, and communication.

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.

Agha describes actors, which extend the concept of objects to concurrent computation (Agha, 1997). 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.

VII. 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 (Davis, *et. al*, 1999). 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 (Lee and Xiong, 2000). 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 (Abu-Ghazaleh, *et. al*, 1998). A model in a domain is called a facet, and heterogeneous models are multifaceted designs (Alexander, 1998).

Particular heterogeneous combinations have also been described. Jourdan *et al.* (1994) have proposed a combination of Argos, a hierarchical finite-state machine language, with Lustre, which has a more dataflow flavor, albeit still within a synchronous/reactive concurrency framework. Another interesting integration of diverse semantic models is done in Statemate (Harel, *et. al*, 1990), 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.

VIII. CONCLUSIONS

Embedded software has particular requirements that make the standard abstractions in software development inadequate. Concurrency and time have to rise to first-class status in the semantic models that are used. There are many good ideas for how to do this, and the next phase is likely to be both a consolidation of these ideas, and bringing them to practice.

REFERENCES

- [1] 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.
- [2] 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.
- [3] P. Alexander, "Multi-Faceted Design: The Key to Systems Engineering," in *Proceedings of Forum on Design Languages (FDL-98)*, September, 1998.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.
- [9] 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.
- [10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, and M. Stal, *Pattern-Oriented Software Architecture – A System of Patterns*, Wiley, 1996.
- [11] 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.
- [12] B. P. Douglass, *Real-Time UML*, Addison Wesley, 1998.
- [13] B. P. Douglass, *Doing Hard Time – Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*, Addison Wesley, 1999.
- [14] C. J. Fidge, "Logical Time in Distributed Systems," *Computer*, Vol. 24, No. 8, pp. 28-33, Aug. 1991.
- [15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.
- [16] 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.
- [17] Yuri Gurevich, "May 1997 Draft of the ASM Guide", University of Michigan EECS Department Technical Report CSE-T R-336-97.
- [18] 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.
- [19] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Sci. Comput. Program.*, vol 8, pp. 231-274, 1987.
- [20] 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.
- [21] 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.
- [22] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A Time-triggered Language for Embedded Programming," <http://www-cad.eecs.berkeley.edu/~tah/Publications/giotto.html>, 2000.
- [23] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8, August 1978.
- [24] R. E. Johnson, "Frameworks = (Components + Patterns)," *Communications of the ACM*, Vol. 40, No. 10, pp. 39-42, October 1997.
- [25] 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.
- [26] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Proc. of the IFIP Congress 74*, North-Holland Publishing Co., 1974.
- [27] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, M. Gonzalez Harbour, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*, Kluwer 1993.
- [28] H. Kopetz, M. Holzmann, W. Elmenreich, "A Universal Smart Transducer Interface: TTP/A," *3rd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'2000)*.
- [29] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, Vol. 21, No. 7, July, 1978.
- [30] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, Reading MA, 1997.
- [31] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, September, 1987.
- [32] E. A. Lee and T. M. Parks, "Dataflow Process Networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773-801, May, 1995.
- [33] E. A. Lee and A. Sangiovanni-Vincentelli, "A Framework for Comparing Models of Computation," *IEEE Transaction on CAD*, December 1998.
- [34] E. A. Lee and Y. Xiong, "System-Level Types for Component-Based Design," Technical Memorandum UCB/ERL M00/8, Electronics Research Lab, University of California, Berkeley, CA 94720, USA, February 29, 2000.
- [35] S. Liao, S. Tjiang, R. Gupta, "An efficient implementation of reactivity for modeling hardware in the Scenic design environment," *Proc. of the Design Automation Conference (DAC 97)*, Anaheim, CA, 1997.
- [36] D. C. Luckham and J. Vera, "An Event-Based Architecture Definition Language," *IEEE Transactions on Software Engineering*, 21(9), pp. 717-734, September, 1995.
- [37] 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.
- [38] R. Milner, "A Theory of Type Polymorphism in Programming," *Journal of Computer and System Sciences* 17, pp. 384-375, 1978.
- [39] 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.
- [40] G. M. Reed and A. W. Roscoe, "A Timed Model for Communicating Sequential Processes," *Theoretical Computer Science*, 58(1/3): 249-261, June 1988.
- [41] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design of the TAO Real-Time Object Request Broker," *Computer Communications*, Elsevier Science, Volume 21, No 4, April, 1998.
- [42] D. C. Sshmidt, M. Stal, H. Rohnert, F. Buschmann, *Pattern-Oriented Software Architecture – Patterns for Concurrent and Networked Objects*, Wiley, 2000.
- [43] 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.