Computing for Embedded Systems

By Edward A. Lee

•Embedded software is increasingly a composition of concurrent components.

•Components in embedded systems interact in a variety of ways, not limited to the simple transfer of control of method calls in object-oriented design.

Models of Computation

•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.

•In embedded software design, components are concurrent processes or threads, typically scheduled by a priority-driven real-time operating system.

Patterns and Frameworks

• *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.

•A *framework* is a set of constraints on components and their interaction, and a set of benefits that derive from those constraints.

An Example of Framework

• One example of framework 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*).

• The key challenge in embedded software research is to invent frameworks with properties that better match the application (e.g. real time requirement).

Architecture Description Languages

•Certain architecture description languages (ADLs), such as *Wright* and *Rapide* define a model of computation.

•But sometimes, some models are hard to describe.(e.g. Wright, which is based on CSP, does not cleanly describe asynchronous message passing.)

•*Architecture design languages* are preferred, whose focus should not be on describing current practice, but rather on improving future practice.

Programming Languages

• It is fairly common to support models of computation with language extensions or entirely new languages.

•An alternative approach is to explicitly use models of computation for coordination of modular programs written in standard, more widely used languages.

Examples of Models of Computation

1. *Continuous time and differential equations*

- The components represent relations between continuous-time functions, and the interactions are the continuous-time functions themselves.
- Differential equations are excellent for modeling analog circuits and many physical systems.
- Joint modeling of a continuous subsystem with digital electronics is known as mixed signal modeling.

Examples of Models of Computation (continued)

2. *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).

Examples of Models of Computation (continued)

3. *State machines*

- In finite state machines (FSMs), components represent system state and the interactions represent state transitions.
- FSM models are not as expressive as the other models of computation described here. And the number of states can easily get very large.
- Combining FSM models with concurrent models of computation to overcome the weakness.

Examples of Models of Computation (continued)

4. *Synchronous/reactive models*

- In the SR model of computation, 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.
- SR models are excellent for applications with concurrent and complex control logic.

Examples of Models of Computation (continued)

5. *Discrete-event models*

- In 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.
- DE models are excellent descriptions of concurrent hardware (telecommunications systems).

Examples of Models of Computation (continued)

6. *Cycle-driven models*

- Cycle driven models associate components with clocks and stimulate computations regularly according to the clock ticks.
- Although discrete-event modeling for such systems is possible, it is costly, primarily due to the priority queue that sorts events chronologically.

Examples of Models of Computation (continued)

7. *Rate monotonic scheduling*

- 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.
- It does not intrinsically include any interaction mechanisms.

Examples of Models of Computation (continued)

8. *Synchronous message passing (Rendezvous)*

- In synchronous message passing, the components are processes, and processes communicate in atomic, instantaneous actions called rendezvous.
- 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.
- CSP is a variant of Synchronous message passing .

Examples of Models of Computation (continued)

9. *Asynchronous message passing*

- In asynchronous message passing, processes communicate by sending messages through channels that can buffer the messages.
- Kahn process network (PN) models and dataflow models are variants of this techniques.
- PN models are excellent for signal processing.
- Synchronous dataflow (SDF) is a particularly restricted special case with the extremely useful property that deadlock and boundedness are decidable.

Examples of Models of Computation (continued)

10. *Timed CSP and timed PN*

- Neither CSP or PN model intrinsically includes a notion of time, which can make it difficult to interoperate with models that do include a notion of time.
- Both CSP and PN models of computation can be augmented with a notion of time to promote interoperability and to directly model temporal properties.

Examples of Models of Computation (continued)

11. *Publish and subscribe*

- The PS model of computation uses notification of events as the primary means of interaction between components.
- 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).

Examples of Models of Computation (continued)

12. *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.
- It has a key advantage that since no synchronization is built in, unsynchronized interactions can be easily implemented with no risk of deadlock.

Choosing Models of Computation

•It is expected that sophisticated and highly visual user interfaces will be needed to enable designers to cope with this heterogeneity of models of computation.

•An essential difference between concurrent models of computation is their modeling of time.

•Two alternatives to choose models (unified or subsuming).

•To design interesting systems, designers need to use heterogeneous models.

Framework Frameworks

•To avoid giving up the benefits of specialized frameworks, designers of these complex systems will have to mix frameworks heterogeneously.

•Mixing frameworks through specialization or hierarchically.

•The approach in the Ptolemy project is to use a system-level type concept that we call domain polymorphism. A component that is domain polymorphic is one that can operate in a number of domains.

Overview of the Ptolemy Project

•The Ptolemy project studies heterogeneous modeling, simulation, and design of concurrent systems for *embedded systems*.

•*Modeling* is the act of representing a system or subsystem formally.

•*Design* is the act of defining a system or subsystem.

•*A major emphasis* in Ptolemy II is on the methodology for defining and producing embedded software together with the systems within which it is embedded.

Overview of Ptolemy II

•*A principle* of the Ptolemy project is that the choices of models of computation strongly affect the quality of a system design.

•*The objective* in Ptolemy II is to support the construction and interoperability of executable models that are built under a wide variety of models of computation.

•*Component-based* design in Ptolemy II involves disciplined interactions between components governed by a model of computation.

Architecture Design

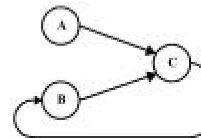
Ptolemy II might be called an *architecture design language*.

•Components are designed to be *domain polymorphic*, meaning that they can interact with other components within a wide variety of domains.

•Ptolemy II provides a rich set of interaction mechanisms embodied in the Ptolemy II domains.

•Ptolemy II has developed a more abstract formal framework that describes models of computation at a meta level.

Models of Computation (Domains)



Ptolemy II models are graphs as above, where the nodes are *entities* and the arcs are *relations*. For most domains, the entities are *actors* and the relations connecting them represent communication between actors.

Models of Computation (Domains)
(continued)

The following models are implemented or planed:

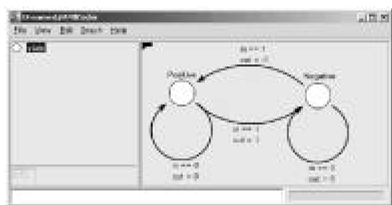
- Communicating Sequential Processes – CSP (Synchronous message passing)
- Continuous Time – CT
- Discrete-Events – DE
- Distributed Discrete Events – DDE

Models of Computation (Domains)
(continued)

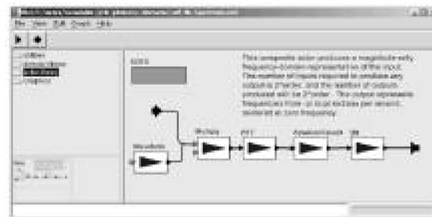
5. Discrete Time – DT
6. Finite-State Machines – FSM (states, not actors)
7. Process Networks – PN
8. Synchronous Dataflow – SDF
9. Synchronous/Reactive – SR (not implemented)

Visual Syntaxes

Visual depictions of systems can help to offset the increased complexity that is introduced by heterogeneous modeling.



Visual Syntaxes
(continued)



Ptolemy II Architecture

•Generic packages that contain math libraries, graph algorithms, an interpreted expression language, signal plotters, and interfaces to media capabilities such as audio.

•Specialized packages includes packages that support clustered graph representations of models, packages that support executable models, and *domains*, which are packages that implement a particular model of computation.

Ptolemy II Architecture
(continued)

Ptolemy II is modular, with a careful package structure that supports a layered approach:

•The *core packages* support the data model, or *abstract syntax*, of Ptolemy II designs.

•The *User Interface(UI) packages* provide support for our XML file format, called MoML, and a visual interface for constructing models graphically.

Ptolemy II Architecture
(continued)

•The *library packages* provide actor libraries that are *domain polymorphic*, meaning that they can operate in a variety of domains.

•And finally, the *domain packages* provide domains, each of which implements a model of computation, and some of which provide their own, domain-specific actor libraries.

Capabilities

•*Higher level concurrent design in Java™.*

•*Better modularization through the use of packages.*

•*Complete separation of the abstract syntax from the semantics.*

•*Improved heterogeneity via a well-defined abstract semantics.*

•*Thread-safe concurrent execution.*

•*A software architecture based on object modeling.*

•*A truly polymorphic type system.*

•*Domain-polymorphic actors.*

Future Capabilities

•*Extensible XML-based file formats*.

•*Interoperability through software components*.

•*Code generation.*

•*Integrated verification tools.*

•*Reflection of dynamics*.

•*Meta modeling*.