Polymorphic Actor-Oriented Design for Heterogeneous Embedded Software

Abstract for HPEC '03

Edward A. Lee, Professor (US citizen)
University of California at Berkeley
518 Cory Hall, Berkeley, CA 94720
(510)642-0455, fax: (510)510-642-2739

1.0 Submission Details

1.1 Session Preferences

Due to travel contraints, I may not be able to stay for the full workshop. It would be safest if this presentation could be scheduled on the first day. It need not be in a US-only session. I would prefer a talk over a poster.

1.2 Areas Addressed

- Networked Embedded Systems
- Software Architecture, Reusability, Scalability, and Standards
- Middleware Libraries and Application Programming Interfaces

2.0 Abstract

Actor-oriented models have been used in distributed computation and embedded real-time software design. The components in such models are conceptually concurrent, and interact via ports, which are producers or recipients of data, rather than via procedure calls, as is common in object-oriented abstractions. Whereas procedures define a specific interaction mechanism (e.g. call-by-reference and transfer of control), ports imply fewer constraints—they represent the fact of an interaction rather than a mechanism of interaction. Ports, for example, might support publish-and-subscribe interactions, push or pull of data, rendezvous, or asynchronous message passing. Although all of these can specified using procedure-based interfaces (in the form of middleware), doing so usually requires specializing the component to the interaction mechanism. When component interfaces are defined in terms of ports, no such specialization is necessary.

Examples of actor-oriented modeling frameworks include Simulink, from The Mathworks, ROOM, which is evolving into a significant portion of UML-2, the Boeing Open Control Platform (OCP), the Navy's PGM, Metropolis, SystemC, and many others. The Ptolemy Project at Berkeley has been exploring actor-oriented methodologies, focusing in

particular on the semantics of actor interaction. One contribution of the Ptolemy Project has been the notion of "domain polymorphism," in which actors (the components of actor-oriented design) are able to operate within more than one semantic framework. This extends the highly successful object-oriented notion of polymorphism, where objects (the components of OO design) are able to operate on multiple data types.

Because of their intrinsic concurrency, actor-oriented frameworks are naturals for distributed software. In such scenarios, the semantics of actor interaction is typically provided by the middleware, using for example the CORBA Event Service for a publish-and-subscribe semantics, or Trader for discovery. However, middleware architectures today define object-oriented, not actor-oriented APIs through which components interact. These APIs leverage the key features of object-oriented design, particularly inheritance and polymorphism, to get data abstraction. This creates a measure of component reusability and adaptability. Many practitioners have, however, been disappointed with the reusability and adaptability achieved in practice, as such components build in many resource management and quality-of-service details that make it difficult to reuse a component in a different context, and make it difficult for a component to adapt to changes in the system architecture.

For example, a distributed component architecture may use a data replication service to optimize effective use of network bandwidth. A different architecture may use the CORBA event service to accomplish a publish-and-subscribe style of component interaction. A third architecture may use Jini to discover a communication service and mobile code that implements a datagram stream. It is very difficult to define components that can interact through more than one of these mechanisms. The API's are different, and the protocols used for interacting with the communication fabric are different. It is arguable that in many cases, inclusion of the details of these protocols in the component design is inappropriate. To change middleware, we face a daunting programming task. Today, the policies and strategies of the middleware affect the design of the components—inappropriately, since they are properly part of the platform through which the components interact, rather than part of the component design. Why should the designer of a text-to-speech component, for example, know about the mechanisms used to ensure the privacy of the transmitted text?

One solution is to design adapters. However, the concept of data abstraction demonstrates that adapter components are not the only solution. Polymorphic interfaces are an alternate solution. The key issue is that data abstraction deals with static structure, and not with the dynamic properties of software. The problem, then, is to identify a way to make components polymorphic with respect to dynamic properties rather than just static structure. How can we design components that will work with a variety of mechanisms for providing input data, so that, for example, we can use these components in a peer-to-peer architecture where data is highly distributed and locally cached, but without building knowledge of the peer-to-peer protocol into the component?

Various approaches begin to address the problem, including for example the use OCL with UML as a meta-programming environment (as done by the Vanderbilt GME) and various OMG efforts such as the model-driven architecture (MDA), Statecharts, the UML "action language" and the definition of "executable UML." To build a solid foundation that deals

with dynamic properties, however, what we need is something more akin to type systems, which lend themselves to rigorous soundness analysis and consistency checking.

The “domain polymorphism” of the Ptolemy Project is based on sound and scalable “interface theories,” where dynamic properties of components are represented as “behavioral types.” These behavioral types can function in design much like classical data types, in that they can be checked statically and dynamically, they are amenable to subclassing and inheritance, and they support polymorphism. But unlike data types, they represent system dynamics.