# DESIGN METHODOLOGY FOR DSP

*Edward A. Lee, Principal Investigator*

Department of Electrical Engineering and Computer Science
University of California, Berkeley CA 94720

Final Report 2001-02, Micro Project #01-048
Industrial Sponsors: Agilent, Cadence, Philips

**ABSTRACT**

The Ptolemy project studies modeling, simulation, and design of concurrent, real-time, embedded systems. The focus is on assembly of concurrent components. The key underlying principle in the project is the use of well-defined models of computation that govern the interaction between components. A major problem area being addressed is the use of heterogeneous mixtures of models of computation. A software system called Ptolemy II is being constructed in Java. The overall Ptolemy project is fairly large, with additional support from DARPA, GSRC, and a number of other companies, and is strongly collaborative. The MICRO project has focused on real-time signal processing, although the larger project is broader.

## 1. The Context

The objectives of the Ptolemy Project include many aspects of designing embedded systems, ranging from designing and simulating algorithms to synthesizing hardware and software, parallelizing algorithms, and prototyping real-time systems. Research ideas developed in the project are implemented and tested in the Ptolemy software environment. The Ptolemy software environment, which serves as our laboratory, is a system-level design framework that allows mixing models of computation and implementation languages.

In designing digital signal processing and communications systems, often the best available design tools are domain specific. The tools must be able to interact. Ptolemy allows the interaction of diverse models of computation by using the object-oriented principles of polymorphism and information hiding. For example, using Ptolemy, a high-level dataflow model of a signal processing system can be connected to a hardware simulator that in turn may be connected to a discrete-event model of a communication network.

A part of the Ptolemy project concerns programming methodologies commonly called "graphical dataflow programming" that are used in industry for signal processing and experimentally for other applications. By "graphical" we mean simply that the program is explicitly specified by a directed graph where the nodes represent computations and the arcs represent streams of data. The graphs are typically hierarchical, in that a node in a graph may represent another directed graph. In Ptolemy II the nodes in the graph are subprograms specified in Java.

It is common in the signal processing community to use a visual syntax to specify such graphs, in which case the model is often called "visual dataflow programming." But it is by no means essential to use a visual syntax.

Hierarchy in graphical program structure can be viewed as an alternative to the more usual abstraction of subprograms via procedures, functions, or objects. It is better suited than any of these to a visual syntax, and also better suited to signal processing.

Some other examples of graphical programming environments intended for signal processing the Advanced Development System (ADS), which is based on Ptolemy Classic, from Agilent, the signal processing worksystem (SPW), from Cadence, CoCentric Design Studio, from Synopsys, and Simulink, from The MathWorks. SPW and CoCentric both use dataflow models that were developed as part of this project.

All of these software environments define applications as assemblies of components that are coordinated in some way. Many possibilities have been explored for precise semantics of the coordination. Many of these limit expressiveness in exchange for considerable advantages such as compile-time predictability. In Ptolemy, a *domain* defines the semantics of the coordination between components. Domains are modular objects that can be mixed and matched at will, thus getting a rich and rigorous approach to heterogeneous modeling.

Graphical programs can be either interpreted or compiled. It is common in signal processing environments to provide both options. The output of compilation can be a standard procedural language, such as C, assembly code for programmable DSP processors, or even specifications of silicon implementations. A major part of the work in the next period will be on such compilation.

## 2. Results of Micro Support

### *2.1. Ptolemy II*

We have built a second generation of design software called Ptolemy II. It is written in Java, is fully network-integrated, is capable of operating within the worldwide web and enterprise software architectures, and is multithreaded.

Ptolemy II offers a unified infrastructure for implementations of a number of models of computation. The overall architecture consists of a set of packages that provide generic support for all models of computation and a set of packages that provide more specialized support for particular models of computation.

Examples of the former include packages that contain math libraries, graph algorithms, an interpreted expression language, signal plotters, and interfaces to media capabilities such as audio. Examples of the latter include 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.

## 2.2. Specification of control flow in Ptolemy

Most modern systems include control logic for the proper sequencing of computational tasks, and for switching and coordinating between different operating modes. Finite state machines (FSMs) have long been used to specify control flow for control-dominated problems. In large systems, however, the control functionality can become so complex that the flat, sequential FSM model becomes impractical. Hierarchical concurrent FSMs (HCFSMs) increase the usefulness of FSMs by extending them with structuring and communication mechanisms. However, most formalisms that support HCFSMs, such as Statecharts and its variants, tightly integrate the concurrency semantics with the FSM semantics. Based on the Ptolemy philosophy of hierarchical composition of heterogeneous models of computation, a formalism called "*charts" (pronounced "starcharts") allows embedding hierarchical FSMs within various concurrency models, in particular continuous time, dataflow, discrete event and synchronous/reactive models. In this heterogeneous model, the semantics of FSM, concurrency and hierarchy are naturally supported. Our scheme decouples the FSM from the concurrency models, enabling selection of the most appropriate concurrency model for the problem at hand.

Xiaojun Liu has developed an FSM domain in Ptolemy II, which is integrated with a number of other domains (such as CT, DE, SDF, and Giotto). We are currently working on integrating the FSM domain with process-based domains such as CSP and PN. We will also investigate what formal verification methods can be applied to the *charts formalism.

## 2.3. Distributed processing in Ptolemy

A large number of embedded systems applications require the coordination of physically separated components, or networked embedded sub-systems. Distributing system components across a network can improve the robustness of a system and simplify its architecture by allowing components to run concurrently and independently. It also facilitates the exploitation of the intrinsically parallel nature of specialized hardware, offering the promise of improved execution speed.

Traditional distributed computing is built on the client-server model, which lacks object-orientation and presents obstacles to scaling it up. Middleware technologies, like the Common Object Request Broker Architecture (CORBA) and the Distributed Component Object Model (DCOM) offer object models and scalability, but the programming model is too liberal to allow any analysis of formal properties of a system. We use Ptolemy's support for the definition of new models of computation to define and implement well-structured models for the interaction of distributed components. The intent is to explore the concept of models of computation in distributed software systems and study the implication of Ptolemy component architecture, message passing mechanisms and execution models.

In an early study, we have demonstrated importing a remote model as a component of a larger model and executing it in a distributed fashion over a network, and implemented a publish/subscribe type of message passing mechanism based on JINI, JavaSpaces and the CORBA Event Service.

In order to support distributed components, three distributed objects -- receivers, parameters, and executable interfaces -- were exported via CORBA. These objects are the basis of Ptolemy II support of models of computations that govern interacting components. Distributed Ptolemy II entities make use of these CORBA objects to pass messages and transfer execution control, obeying the model of computation defined by the director, regardless of their locations on the network.

In this report period, Yang Zhao and Xiaojun Liu implemented a new domain in Ptolemy II in order to study the communication behavior, resource management and model partitioning among distributed components or sub-systems. By decoupling the communication behavior from the computation behavior of a component or sub-system, we can achieve higher modularity, and better component reuse.

Current work includes the definition of a clean interface for distributed components and their interaction, and the exploration of real-time issues under the framework. We also plan to formalize the CI domain and refine it with more features, such as time, priority, etc.

## 2.4. A model of computation for networked embedded systems

Networked embedded systems such as wireless sensor networks are usually designed to be event-driven so that they are reactive and power-efficient. Programming embedded systems with multiple reactive tasks is difficult due to the complex nature of managing the concurrency of execution threads and consistency of shared states. Elaine Cheong collaborated with Judith Liebman (UC Berkeley), and Jie Liu and Feng Zhao (at the Palo Alto Research Center) in designing and implementing a globally asynchronous and locally synchronous model, called TinyGALS, for programming event-driven embedded systems. Software components are composed locally through synchronous method calls to form modules, and asynchronous message passing is used between modules to separate the flow of control. In addition, we have designed a guarded yet synchronous model, TinyGUYS, which allows thread-safe sharing of global state by multiple modules without explicitly passing messages. Our notions of synchrony and asynchrony, which are consistent with the usage of these terms in distributed programming paradigms, refer to whether the software flow of control is immediately transferred to a component.

With this highly structured programming model, all asynchronous message passing code and module triggering mechanisms can be automatically generated from a high-level specification. The programming model and code generation facilities have been implemented for a wireless sensor network platform known as the Berkeley motes. TinyOS is an event-based operating system for these networked sensors being developed by the group of Prof. David Culler. Our implementation of TinyGALS uses the component model provided by TinyOS, which has an interface abstraction that is consistent with synchronous method calls. The TinyGALS code generator is designed to work with preexisting TinyOS components, thus enabling code reuse.

The key contribution of TinyGALS is in the way developers can view the system they are creating. Developing such a highly concurrent embedded system in a traditional programming language leads to a structure that is oriented along functionally similar entities and the messages they exchange (see figure 1), but it reveals nothing about the relation between these messages, the common access to global state, or the various concurrent threads running in the system.

These aspects are made explicit in a TinyGALS model of the same system (see figure 2). It directly represents threads and global state, and it makes explicit the dependencies between the signals received and set by a component.

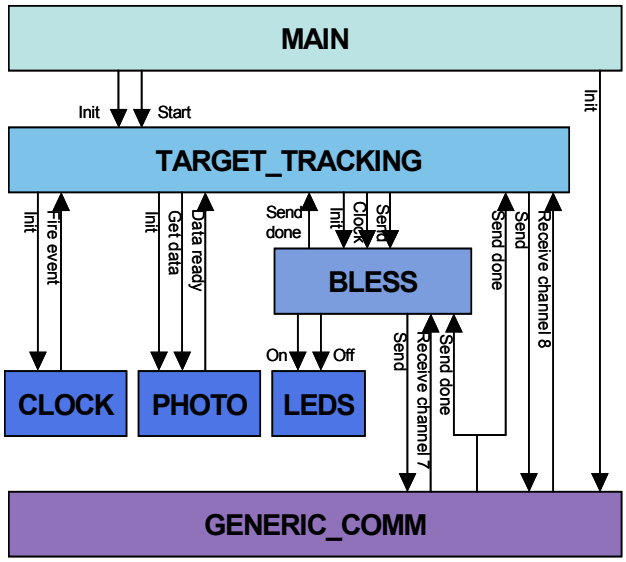


**Figure 1. Traditional structure.**

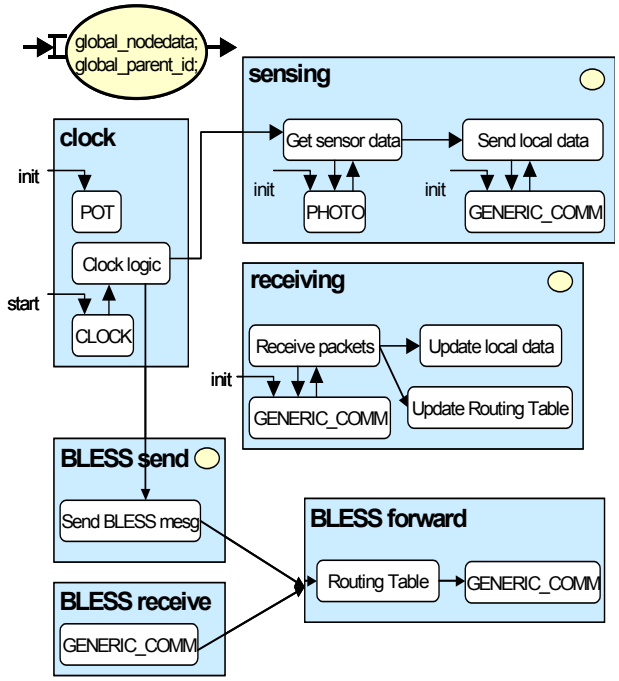


**Figure 2. TinyGALS structure.**

We are currently working on formalizing the dynamic properties of TinyGALS, as well as developing a more robust implementation for communication and scheduling. TinyGALS is currently intended for running on a single node. We plan to extend the TinyGALS model to multiple nodes for distributed multi-tasking.

## *2.5. Actor language*

Ptolemy II provides a sophisticated software infrastructure for the development of actor-based models, as well as definition of individual actors themselves. However, defining actors directly in Ptolemy's host language (Java) requires a degree of familiarity with this infrastructure that might deter many more casual users of the system. The CAL actor language was designed to address this issue, allowing users to express actors in a small domain-specific language.

Johan Eker (now at Ericsson), Ed Willink (at Thales Research), and Jörn W. Janneck have collaborated on the language design and on tools for parsing, analyzing, and transforming actors in that language. Lars Wernli has contributed a code generator translating actors to Java so that they can be run inside the Ptolemy II framework. Johan Eker and Yang Zhao have written an experimental C code generator.

## *2.6. Code generation*

The high-level of abstraction possible in component-based modeling offers many advantages, such as simulation speed, the strength of formal models of computation, etc. However, the fundamental weakness of high-level modeling is the difficulty of actual implementation. Traditionally the easiest way to get high performance has been to translate the model by hand into a low-level implementation language. Automatic code generation from the model is sometimes possible by assembling the component specifications, but only with serious performance penalties. These penalties come from several sources:

- A component in the modeling environment is inevitably built to be easy to design with. Components can often accept a variety of data types (type-polymorphism), can be used in a variety of control or communication models (domain-polymorphism), can be configured with different parameters (operational-polymorphism), and can be used in a variety of environments (context-polymorphism). This flexibility directly conflicts with the goals of most optimized implementations.
- A model consists of components, their ports, and the connections between those ports. The model of computation associated with a model determines how connected components communicate and control their execution. In some cases the boundaries of components correspond to physical boundaries of the implemented system. For example, there might be one component for each physical processor that is connected to a system bus. However, the boundaries of components often have no physical importance in a system. They are specified arbitrarily by the system designer, or because of the social structure of group designing the model, or because of the availability of reusable components. Simple code generation strategies blindly preserve the structure of the model, which can result in unnecessary overhead.

At some level, these problems can be ameliorated using traditional code generation strategies. If a component is too flexible to

generate good code, then it can be replaced with a specialized hand-written version. If there is unnecessary structure in the model, then change the model so that the structure is removed. However, these solutions conflict directly with good engineering practice, and greatly complicate the implementation procedure. Steve Neuendorffer and Christopher Hylands collaborate on developing a code generation strategy that attempts to handle these difficulties automatically. The key idea is to combine code generation from a model with compilation of the code for individual actors. We call this strategy *Co-compilation*. This strategy directly addresses the difficulties above. We parse the code for an actor and specialize it according to its use in a particular model (the types, the particular domain, the values of parameters and the connections that have been made to it). We can also perform cross-actor optimizations to eliminate or reorganize the structure of a model.

Co-compilation also offers a straightforward path to code generation from heterogeneous models that contain different communication and control strategies organized in a hierarchical structure. We anticipate being able to generate code for a model at one level of the hierarchy and then use the generated code as a component at a higher level of the hierarchy. This can result in reduced overhead as well, since a system designer is not limited to a single model of computation.

We have implemented this code-generation strategy as the Copernicus package, which is part of Ptolemy II. Copernicus parses the Java bytecode for actors, optimizes it, combines it with code generation from the model and outputs Java bytecode. The resulting generated code is currently useful for high-speed compiled code simulation. We are currently exploring how to generated code for embedded architectures and for FPGAs.

### *2.7. Status*

In this report period a new major version of the Ptolemy II software was released (version 2.0.1 in August 2002). It includes a limited prototype of our code generation facility that will generate class files for non-hierarchical SDF models, as well as support for modal models, a Timed Multitasking domain and a Synchronous reactive domain.

During this report period HyVisual, a specialized version of the Ptolemy II software, focussing on the description of hybrid systems, was prepared for release. It shipped in January 2003.

## 3. Publications

This project has generated a number of publications during this reporting period. Here are some of the highlights.

### *3.1. Journal Articles*


[1] Edward A. Lee, "Embedded Software," Advances in Computers (M. Zelkowitz, editor), Vol. 56, Academic Press, London, 2002

[2] Jozsef Ludvig, James McCarthy, Stephen Neuendorffer, Sonia R. Sachs,"Reprogrammable Platforms for High-Speed Data Acquisition," Kluwer Journal of Design Automation for Embedded Systems, Volume 7, Number 4, November, 2002

[3] Praveen K. Murthy and Edward A. Lee, "Multidimensional Synchronous Dataflow," IEEE Transactions on Signal Processing, volume 50, no. 8, pp. 2064 -2079, August 2002.


### *3.2. Conference Papers*


[4] Jie Liu, Johan Eker, Jorn W. Janneck and Edward A. Lee, "Realistic Simulations of Embedded Control Systems," Proceedings of the 15th IFAC World Congress, Barcelona, Spain, July 21-26, 2002


### *3.3. Ph.D. Dissertations*


[5] Yuhong Xiong,"An Extensible Type System for Component-Based Design," Ph.D. thesis, Technical Memorandum UCB/ERL M02/13, University of California, Berkeley, CA 94720, May 1, 2002


### *3.4. Masters Reports*


[6] Stephen Neuendorffer, "Automatic Specialization of Actor-Oriented Models in Ptolemy II," Master's Report, Technical Memorandum UCB/ERL M02/41, University of California, Berkeley, CA 94720, December 25, 2002.


### *3.5. Other Technical Reports*


[7] J. Adam Cataldo, Edward A. Lee, and Xiaojun Liu, "Preliminary Version of a Two-Dimensional Technical Specification for Softwalls," Technical Memorandum UCB/ERL M02/9, University of California, Berkeley, CA 94720, April 17, 2002.

[8] Jörn W. Janneck, "Actors and their composition," Memorandum UCB/ERL M02/37, University of California at Berkeley, 18 December 2002.

[9] Edward A. Lee and Yuhong Xiong, "Behavioral Types for Component-Based Design," Memorandum UCB/ERL M02/29, University of California, Berkeley, CA 94720, USA, September 27, 2002

[10] H. John Reekie and Edward A. Lee, "Lightweight Component Models for Embedded Systems," Memorandum UCB/ERL M02/30, University of California, Berkeley, CA 94720, USA, October 30, 2002.

[11] Lars Wernli, "Design and implementation of a code generator for the CAL actor language," Technical Memorandum UCB/ERL M02/5, University of California, Berkeley, CA 94720, March 2002.