DESIGN METHODOLOGY FOR DSP

Edward A. Lee, Principal Investigator

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

Final Report 1999-00, Micro Project #99-068
Industrial Sponsors: Cadence, Hewlett-Packard, Hughes, 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 dataflow 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. These software environments all claim variants of dataflow semantics, with SPW and CoCentric both using 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. *Visual Syntaxes*

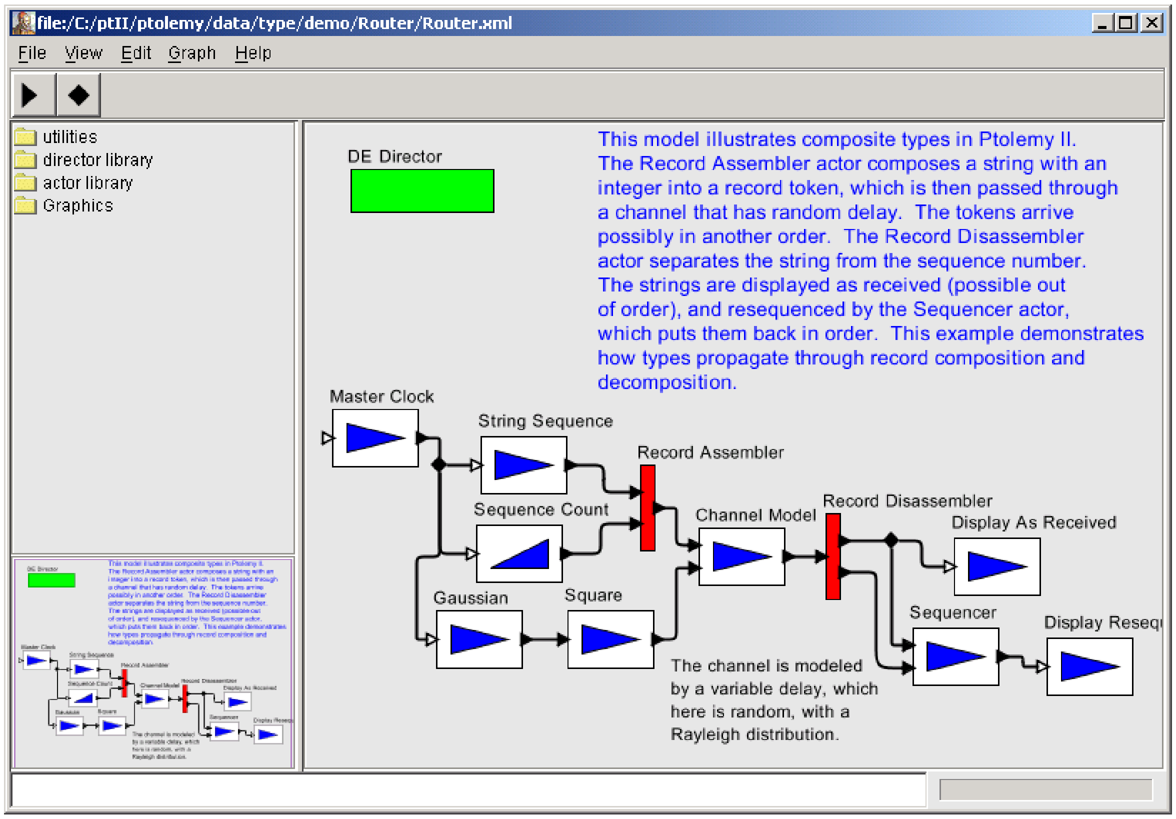
Visual depictions of systems have always held a strong human appeal, making them extremely effective in conveying information about a design. Many of the domains of interest in the Ptolemy project use such depictions to completely and formally specify models. One of the principles of the Ptolemy project is that visual depictions of systems can help to offset the increased complexity that is introduced by heterogeneous modeling.

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.

Figures 1 and 2 show two different visual renditions of Ptolemy II models. Both renditions are constructed in Vergil, the visual editor framework in Ptolemy II. In figure 1, a Ptolemy II model is shown as a block diagram, which is an appropriate rendition for many discrete event models. In this particular example, records are constructed at the left by composing strings with integers representing a sequence number. The records are launched into a network that introduces random delay. The records may arrive at the right out of order, but the Sequence actor is used to re-order them using the sequence number.

Figure 2 also shows a visual rendition of a Ptolemy II model, but now, the components are represented by circles, and the connections between components are represented by labeled arcs. This visual syntax is a familiar way to represent finite state machines (FSMs). Each circle represents a state of the model, and the arcs represent transitions between states. The particular example in the figure comes from a hybrid system model, where the two states, Separate and Together, represent two different modes of operation of a continuous-time system. The arcs are labeled with two lines, the first of which is a *guard*, and the second of which is an *action*. The guard is a boolean-valued expression that specifies when the transition should be taken, and the action is a sequence of commands that are executed when the transition is taken.

The visual renditions in figures 1 and 2 are both constructed using the same underlying infrastructure, Vergil, built by Stephen Neuendorffer. Vergil, in turn, in built on top of a GUI package called Diva, developed by John Reekie and Michael Shilman at Berkeley. Diva, in turn, is built on top of Swing and Java 2D, which are part of the Java platform from Sun Microsystems. In Vergil, a visual editor is constructed as an assembly of components in a Ptolemy II model. Thus, the system is config-



This model illustrates composite types in Ptolemy II. The Record Assembler actor composes a string with an integer into a record token, which is then passed through a channel that has random delay. The tokens arrive possibly in another order. The Record Disassembler actor separates the string from the sequence number. The strings are displayed as received (possible out of order), and resequenced by the Sequencer actor, which puts them back in order. This example demonstrates how types propagate through record composition and decomposition.

The channel is modeled by a variable delay, which here is random, with a Rayleigh distribution.

Figure 1. Visual rendition of a Ptolemy II model as a block diagram in Vergil (in the DE domain).

urable and customizable, and a great deal of infrastructure can be shared between the two distinct visual editors of figures 1 and 2.

A subset of visual languages that are recognizable as "block diagrams" represent concurrent systems. 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 system specification and design. Ptolemy II supports exploration of the possible concurrency semantics. A principle of the project is that the strengths and weaknesses of these alternatives make them complementary rather than competitive. Thus, interoperability of diverse models is essential.

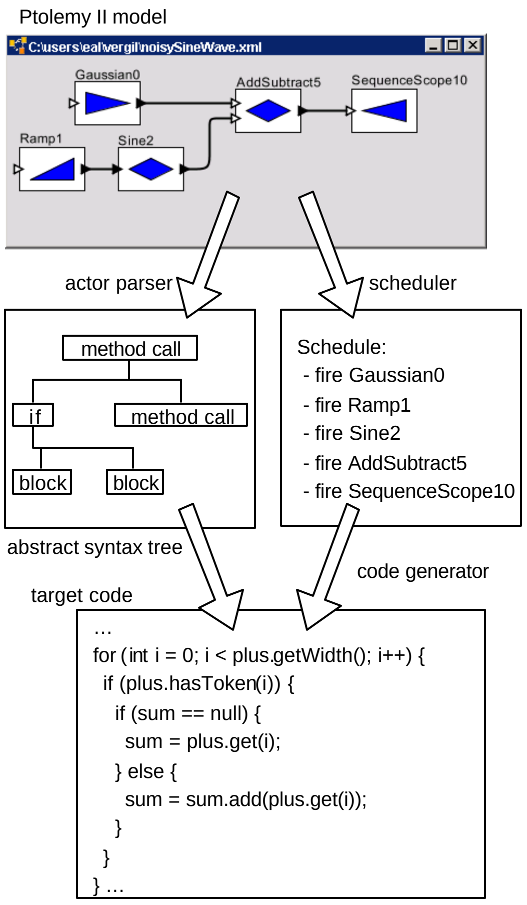
2.3. *Status*

At the end of this project we released the beta version of Ptolemy II 1.0, which includes the Vergil user interface, a number of domains, and an extensive actor library.

3. Code Generation

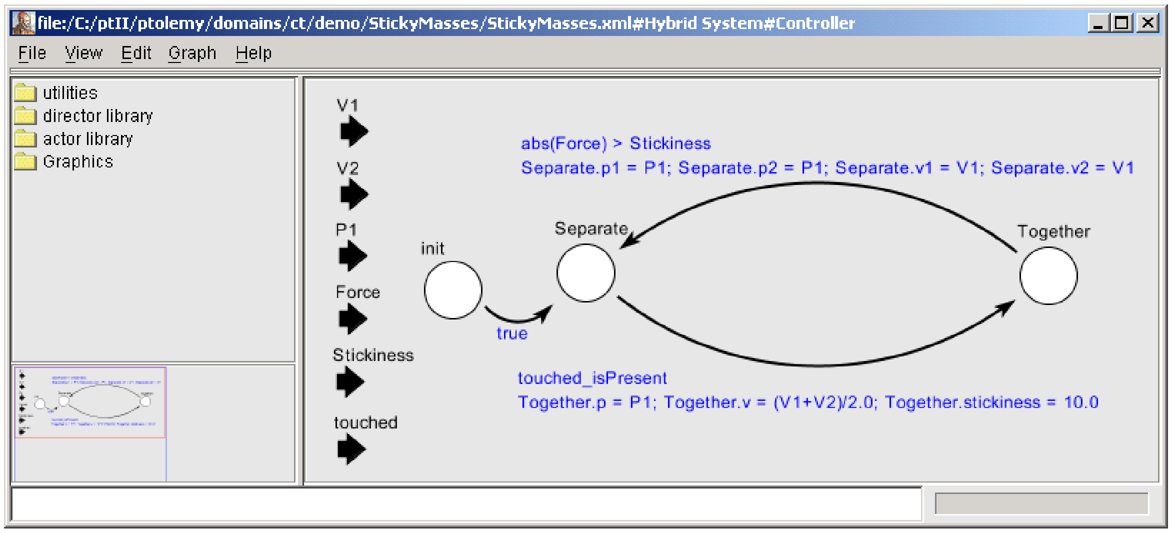
We have begun to make major progress on compiling Ptolemy II models for efficient execution on embedded processors. Jeff Tsay, in a masters project [8], did a pilot project that demonstrates the concept we are following. The approach has elements of a traditional compiler, but a major difference. Figure 3 outlines the approach. The "source code," shown at the top, is a block diagram defined within one of the Ptolemy II domains. It is an assembly of components (called "actors") that are interconnected, where the meaning of the interconnection is determined by the domain semantics.

The approach is to parse the Java code for the actors and construct an abstract syntax tree (AST) for each actor. The AST presents a simple API, using the Visitor pattern, for writing code transformers and back-end code generators, as would be found in a traditional compiler. However, our approach uses the component architecture (the block diagram and domain semantics), which offer information about concurrency, flow of control, and type dependencies that are not available to a traditional compiler. Our approach, thus, is to blend the traditional compiler



```
…
for (int i = 0; i < plus.getWidth(); i++) {
  if (plus.hasToken(i)) {
    if (sum == null) {
      sum = plus.get(i);
    } else {
      sum = sum.add(plus.get(i));
    }
  }
} …
```

Figure 3. Outline of the code generation process.



abs(Force) > Stickiness
Separate.p1 = P1; Separate.p2 = P1; Separate.v1 = V1; Separate.v2 = V1

touched_isPresent
Together.p = P1; Together.v = (V1+V2)/2.0; Together.stickiness = 10.0

Figure 2. Visual rendition of a Ptolemy II model as a state transition diagram in Vergil (FSM domain).

approaches with novel techniques that operate at the component architecture level.

Some generic optimizations, such as specialization of polymorphic data types (leveraging the sophisticated type system in Ptolemy II), have been implemented. Some domain-specific optimizations, such as static buffer allocation for communication between dataflow actors, have been implemented for the synchronous dataflow (SDF) domain, although we plan to also support other Ptolemy II domains.

Along the right path in figure 3, the domain semantics is used to analyze the component architecture and construct schedules (if appropriate to the domain) and generate run-time code supporting domain execution. The back end resynthesizes the transformed Java code, stitched together as needed according to the schedule and/or run-time support code.

There are a number of applications for this infrastructure. First, it can be used simply to transform one Java program into a leaner realization (faster and smaller). Second, it can be used to transform a Java program into an embedded version, for execution on a much smaller virtual machine. We have demonstrated both of these applications with simple examples [2]. More interestingly, embedded C or synthesizable VHDL could be generated to produce highly optimized implementations of the models. Moreover, within Ptolemy II, there are many more domains besides SDF that might benefit from code generation, so that code generated systems may be run without the Ptolemy II software infrastructure in memory or performance constrained scenarios.

The code generator for SDF might be modified to generate code executable in parallel processing environments, if a suitable parallel scheduler were used instead of the existing scheduler. The modifications to the code generator would entail generating a main() method that executes actors in parallel, and modifying certain buffer accesses to block, waiting for other execution paths to complete.

Our plan over the next year is to improve the code generator infrastructure so that it can serve as a laboratory for experimenting with these approaches, and then to do the experimentation.

4. Publications

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

4.1. Journal Articles

[1] Edward A. Lee, "What's Ahead for Embedded Software?," *IEEE Computer*, September 2000, pp. 18-26.

4.2. Conference Papers

[2] Jeff Tsay, Christopher Hylands and Edward Lee, "A Code Generation Framework for Java Component-Based Designs," *CASES '00*, November 17-19, 2000, San Jose, CA.

[3] Jie Liu and Edward A. Lee, "Component-based Hierarchical Modeling of Systems with Continuous and Discrete Dynamics," *Proc. of the 2000 IEEE International Conference on Control Applications and IEEE Symposium on Computer-Aided Control System Design* (CCA/CACSD'00), Anchorage, AK, September 25-27, 2000. pp. 95-100.

[4] Yuhong Xiong and Edward A. Lee, "An Extensible Type System for Component-Based Design," *6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Germany, March/April 2000 . LNCS 1785.

[5] J. Liu, X. Liu, T. J. Koo, B. Sinopoli, S.Sastry, and E. A. Lee, "Hierarchical Hybrid System Simulation," *38th IEEE conference on Decision and Control*, Dec. 1999, Phoenix, AZ.

4.3. Ph.D. Dissertations

[6] John Davis II, "Order and Containment in Concurrent System Design," Ph.D. thesis, Memorandum UCB/ERL M00/47, Electronics Research Laboratory, University of California, Berkeley, September 8, 2000.

[7] Bilung Lee, "Specification and Design of Reactive Systems", Ph.D. thesis, Memorandum UCB/ERL M00/29, Electronics Research Laboratory, University of California, Berkeley, May, 2000.

4.4. Masters Reports

[8] Jeff Tsay, "A Code Generation Framework for Ptolemy II," ERL Technical Report UCB/ERL No. M00/25, Dept. EECS, University of California, Berkeley, CA 94720, May 19, 2000.

4.5. Other Technical Reports

[9] Edward A. Lee, “Embedded Software - An Agenda for Research,” ERL Technical Report UCB/ERL No. M99/63, Dept. EECS, University of California, Berkeley, CA 94720, December 15, 1999.

[10] Edward A. Lee and Steve Neuendorffer, "MoML - A Modeling Markup Language in XML, Version 0.4," Technical Memorandum UCB/ERL M00/12, University of California, Berkeley, CA 94720, March 14, 2000.

[11] Edward A. Lee and Yuhong Xiong, "System-Level Types for Component-Based Design, "Technical Memorandum UCB/ERL M00/8, Electronics Research Lab, University o f California, Berkeley, CA 94720, USA, February 29, 2000.