DESIGN AND SIMULATION OF HETEROGENEOUS CONTROL SYSTEMS USING PTOLEMY II

Johan Eker, Chamberlain Fong, Jörn W. Janneck, Jie Liu

Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, CA 94720-1770, US
{*johane,chf,janneck,liuj*}*@eecs.berkeley.edu*

Abstract: Complex control systems are heterogeneous from both an implementation and a modeling perspective. Design and simulation environments for such systems need to integrate different component interaction styles, like differential equations, discrete events, state machines, dataflow networks, and real-time scheduling. This paper motivates the use of Ptolemy II software environment for modeling and simulation of heterogeneous control systems. Ptolemy II advocates a component-based design methodology, and hierarchically integrates multiple models of computation, which can be used to capture different design perspectives. A Furuta pendulum control system is used as a motivating example. After designing a three-mode hybrid controller under idealized assumptions, implementation effects, like real-time scheduling and network protocols, are taken into consideration to achieve a more realistic simulation. The 3D animation package in Ptolemy II helps designers to visualize the control results. In this process of refining the design, components modeled in early phases can be reused.

Keywords: Simulators, real-time computers, embedded systems

1. INTRODUCTION

The heterogeneity of modern embedded control systems puts high demands on design and simulation tools. A typical hybrid control system consists of a set of subcontrollers and some switching logic. The controllers themselves are conveniently described using discrete equations, the switching logic may be expressed using state machines, the controller is implemented as a task under a real-time operating system, the controlled plant is modeled as differential equations, and so on.

The above subsystems differ in the way their components interact, both in terms of the way they exchange information, and in terms of the flow of control between them—synchronous or asynchronous, buffered or unbuffered, sequential or parallel, etc. Ideally, a designer should be able to model each of these in their natural computational *domain*, based on a corresponding *model of computation*.

This requirement makes the Ptolemy II environment an interesting candidate for creating such models of heterogeneous control systems. Ptolemy II provides a wide (and extensible) range of computational models which are based on a common notion of “domain”. Sub-models in different domains may be hierarchically combined, which makes the simulation model much clearer and more understandable, and facilitates the modeling of complex systems without the need for an equally complex simulation configuration.

Typically, a controller design project proceeds in several distinct phases: modeling, design, simulation and implementation. Clearly, if possible it is preferable to use the same tool through all of the above phases and reuse the components designed earlier. The extensive component libraries and hierarchical compositionality of heterogeneous models make Ptolemy II suitable for refining designs through these phases. The 3D graphical animation support allows the designer to visualize the simulation, and the retargetable code generation facility supports the implementation phase by producing e.g. Java or C code for a particular embedded control system.

This paper shows the application of these concepts and the Ptolemy II environment to the design of a heterogeneous controller by first introducing a small example problem (Section 2) to motivate the Ptolemy approach, and then presenting the pertinent modeling concepts and domains provided by Ptolemy II (Section 3). In Section 4 these concepts are then applied to the example.

2. MOTIVATING EXAMPLE

The inverted pendulum is a classic control problem basically for two reasons: it is nonlinear and unstable. A picture of the inverted pendulum is shown in Fig. 1. The pendulum consists of two moving parts, the arm that moves in the horizontal plane and the pendulum that moves in the vertical plane.

Fig. 1. The rotating inverted pendulum.

A hybrid controller can be designed to swing-up and stabilize the pendulum. The controller consists of three modal subcontrollers: a *swing-up* controller, a *catch* controller, and a *stabilizing* controller. Initially, the pendulum starts in the downward position and the swing-up mode is used to bring it to the top position. Once it is sufficiently close to the top equilibrium, the controller enters the catch mode. The task of the catch mode is to reduce the speed of the pendulum and the arm before the third mode, stabilize, is entered. The catch and stabilize modes are linear state feedback controllers. The swing-up controller uses a nonlinear energy based algorithm for bringing the pendulum to upward position. The controller switching logic can be described using a finite state machine, where each state corresponds to a control mode, and each of the subcontrollers can be described as a dataflow computation, as shown in Fig. 2.

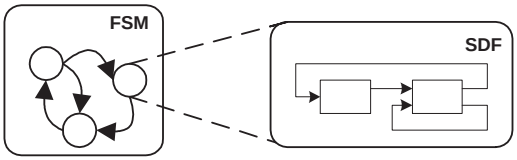

Fig. 2. The hybrid pendulum controller consisting of three subcontrollers.

The above system is straightforward to model and simulate in many modern design tools. One example of it is shown in Fig. 3, where the controller in Fig. 2 is integrated with the plant dynamics, which is modeled as ordinary differential equations (ODEs). However, this

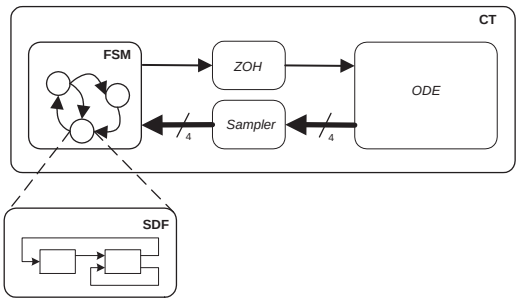

Fig. 3. A basic controller model.

model does not capture many issues related to an actual implementation. Here we make the usual assumptions that the execution time is negligible and that we have no computation and communication jitter. Of course, this is not the case in the real-world. When the controller is running on a real computer and on top of a real-time operating system (RTOS), it will compete with other tasks for resources, e.g. the CPU and I/O. This will give rise to input-output delays and variations in the sampling period. Furthermore, the actuators and the sensors are usually not directly connected to the controller, but instead some network is used for transferring data. The network is a common resource possibly shared by many other control loops, and again the loops compete for the network bandwidth. We would like to capture the above properties so that we can predict the real behavior of the embedded system, and evaluate scheduling mechanisms and communication protocols in terms of applications performance.

In this process of refining a design, designers need to gradually add design considerations to the existing model and migrate the control system from algorithms to implementation. Different design perspectives usually imply heterogeneous component interaction styles. It is desirable that a design environment can support multiple component interaction styles and the components designed in earlier phases can be reused under new interaction styles, so that the verified properties can be preserved as much as possible. We argue that integrating different models of computation will help decompose design perspectives and achieve elegant and reusable models.

3. PTOLEMY II COMPONENT ARCHITECTURE AND MODELS OF COMPUTATION

The Ptolemy II modeling and design environment addresses the heterogeneity and design migration issues using a component-based design methodology, disciplined component interactions, and component reuse. In Ptolemy II, we acknowledge the existence of well-defined models of computation, which guide the communication style and the execution of a composition of components. Each model of computation asserts certain properties that could be desirable for certain aspects of a system. These heterogeneous models of computation can be composed hierarchically to preserve understandability, manage complexity, and encourage component reuse. This section presents the Ptolemy II component structure and the models of computation that are useful for simulation and visualization of control system designs.

3.1 *Model structure*

In Ptolemy II, a model is a hierarchical aggregation of components, which are called *actors*. Actors encapsulate an atomic execution and provide communication interfaces (called *ports*) to other actors. An actor can be *atomic*, meaning that it is at the bottom of the hierarchy. An actor can be *composite*, meaning that it contains other actors. A composite actor can be contained by another composite actor, so hierarchies can be arbitrarily nested.

A port of an actor can be an input, output, or both. Communication channels among actors are established by connecting ports. A port for a composite actor can have connections both to the inside and to the outside, and thus mediates the communication from inside actors to outside actors.

A model of computation associated with a composite actor is implemented as a *domain*. A domain defines the communication semantics and the execution order among actors. The communication mechanism is implemented using *receivers*. To ensure a clear model of computation at a level of hierarchy, all receivers within one composite actor are the same. Receivers could represent FIFO queues, mailboxes, proxies for a global queue, or rendezvous points. Actors, when resident in a domain, adopt domain-specific receivers. By separating computation and communication in this way, many actors can be reused in different domains.

Actors, both atomic and composite, are executable. The execution order of the actors contained in a composite actor is controlled by a *director*. Since receivers and directors must work together, the director is also responsible for creating receivers. When a composite actor is executed, the director associated with the composite actor executes the actors of the contained model. The directors are carefully designed so that they both respect the model of computation they implement and provide a polymorphic execution interface to the outside domain. This compositionality of execution is the key to managing heterogeneity hierarchically.

3.2 *Domains*

A wide variety of domains have been implemented in Ptolemy II. We discuss here a subset of them, which are useful for designing control systems.

3.2.1. *Continuous time* The continuous time (CT) domain (Liu, 1998) models ordinary differential equations (ODEs), extended to allow the handling of discrete events. Special actors, which represent *integrators*, are connected in feedback loops in order to represent the ODEs. Event generators, e.g. periodic samplers, triggered samplers, and zero crossing detectors, and waveform generators, like a zero order hold, are implemented to convert between continuous-time signals and discrete events.

The execution of a CT model involves the computation of a numerical solution to the ODEs at a discrete set of time instance. In order to support the detection of discrete events and interact with discrete models of computation, the time progression and the execution order of a CT model are carefully controlled (Liu and Lee, 2000).

3.2.2. *Discrete event* In the discrete event (DE) domain of Ptolemy II, actors communicate through events placed on a (continuous) time line. Each event has a value and a time stamp. Actors process events in chronological order. The output events produced by an actor are required to be no earlier in time than the input events that were consumed. In other words, DE models are *causal*.

Discrete event models, having the continuous notion of time and the discrete notion of events, are suitable for modeling hardware and software timing properties, communication networks, and queuing systems. A good reference on discrete event systems is (Cassandras, 1993).

3.2.3. *Dataflow models* By “dataflow" we refer to a class of models that interact using asynchronous message passing (Lee and Parks, 1995). In these models, components are processes which communicate by sending messages through FIFO queues. The sender of a message need not wait for the receiver to be ready to receive the message. Most dataflow models only specify the data dependency among components, imposing only a partial ordering on the execution of components. These properties match them well with signal processing and control algorithms, and allow highly efficient execution.

Synchronous dataflow (SDF) (Lee and Messerschmitt, 1987) is a particularly restricted special case of dataflow models with extremely useful properties. In SDF, whenever a component executes, it consumes a fixed amount of input data and produces a fixed amount of outputs. For a consistent SDF model, a schedule can be computed, such that the components do not have to test for sufficient data before execution. So, for algorithms with a fixed structure, SDF is very efficient and predictable.

3.2.4. *Finite State Machines* Finite state machine (FSM) models, see (Hopcroft and Ullman, 1979), are used at two levels, within a component or across components. Within a component, state machines specify precise sequencing of component states and transitions among them. When used across components, state machines can be used to coordinate other models. The typical use is to specify operation modes and sequences among components. These components can be built using other domains (Girault et al., 1999). Hierarchically combining state machines with concurrent models makes state machines concurrent and helps to prevent the explosion of the number of states in complex systems.

3.2.5. *RTOS* The real-time operating system (RTOS) domain in Ptolemy II allows designers to explore priority-based scheduling policies and their effects on real-time software. In this domain, components are software tasks with priorities. The director of this domain implements a prioritized event dispatching mechanism and invokes tasks according to their feasibility and priority. Both preemptive and nonpreemptive scheduling, as well as static and dynamic priority assignment, can be captured.

3.2.6. *3D visualization/graphics* The graphics (GR) domain (described in more detail in (Fong, 2001)) provides a component-based infrastructure for displaying three-dimensional graphics in Ptolemy II. The GR domain follows loop-free synchronous semantics. The basic idea behind the GR domain is to arrange geometry and transform actors in a directed acyclic graph to represent the location and orientation of objects in a world scene. This topology of connected GR actors forms what is commonly known in computer graphics literature as a *scene graph*. Fig. 4 shows an example scene graph topology in Ptolemy II. The geometry actors are source actors that produce basic 3D shapes such as spheres, cylinders, and boxes. The transform actors perform linear transformations on these shapes in order to place them properly in the world scene. Hierarchically combining graphics subsystems with other domains in Ptolemy II is useful for producing animated simulations. The availability of 3D graphics complements Ptolemy's heterogeneous modeling and simulation capabilities, facilitating visualization and understanding of dynamic model behavior.

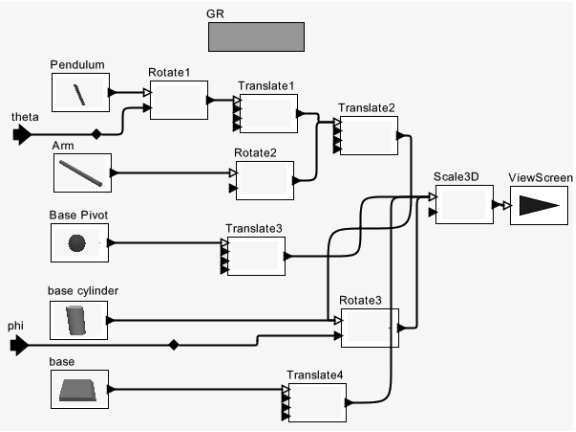

Fig. 4. GR model of the pendulum.

4. REFINING THE MODEL

In this section we will show how a much detailed design of the pendulum control system can be constructed from a basic model using hierarchical composition of domains. The controller is modeled as running on top of a RTOS model, while the sensor and the actuator are distributed and communicate with the controller over a network.

4.1 *Basic model*

The initial model described in section 2 is constructed using three domains in Ptolemy II. The pendulum is modeled in the *continuous time* domain using differential equations. The switching logic of the hybrid controller is described as a *finite state machine*, whose states refine to subcontrollers. These subcontrollers are in turn modeled in the *synchronous dataflow* domain. Here, we assume that the controller takes no time to execute.

4.2 *Model refinement*

This model does not reflect the effects introduced in a concrete implementation, such as controller latency and communication delay. A more accurate model would include a model of the real-time operating system and the network. This is done in two steps in Ptolemy II. First, to consider the real-time issues, we embed the controller designed in the basic model (i.e. the composite actor that contains the finite state machine and the subcontrollers) into an RTOS domain to capture the effects of the interaction between the different tasks running concurrently on the system.

The composite actor for the modal controllers is treated as a task in the RTOS domain. Some other tasks are added to capture the dynamics of concurrent tasks running on the same real-time kernel. Notice that the composite actor we built in the basic model only specifies the computational part of the controller. To actually reflect the implementation, another task, which models the I/O part of the controller, is added, similarly to the task model used in (Eker and Cervin, 1999). This I/O task may compete for resources with other I/O operations running on the system.

The next step is to include a model of the network communication. This is done by using a discrete event domain at the top level, and introducing a network actor, which models the behavior of a given network protocol. We connect both the pendulum model and the controller to the network actor so that the event going through the network will be delayed according to the load of the network. This extended model is shown in Fig. 5. The top-level domain has been changed to DE and it now contains five actors: the pendulum, the controller, the network, an actor that represents other network components, and a visualization actor. Much of the pendulum composite actor reusing the previous pendulum model, which is implemented in the CT domain. The controller is implemented using the RTOS domain described above. The network actor simulates the communication between the nodes in the control system. Arbitrarily many nodes may be connected to the network. The basic functionality of the network actor is to simulate the network behavior, i.e. model multi-node network access, packet dropping, and message latency. The animation composite actor is modeled in the GR domain, and explained further in the next section.

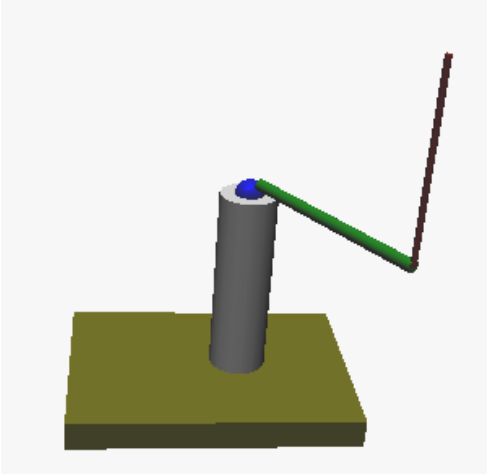

Fig. 6. A 3D model of the Furuta pendulum.

4.3 *Model animation*

To visualize the Furuta pendulum system, we model its core 3D shape in the GR domain. A lofted shape is used to represent the base that supports the moving parts. Cylinders of various cross-sections are used to represent the rotational base, the arm, and the pendulum tip. After translating and rotating these shapes into the right place, we connect them into a scene graph actor. The GR scene graph diagram is shown in Fig. 4. This scene graph specifies that the motions of the rotational base are inherited by the arm and pendulum, and that the motions of the arm are inherited by the pendulum. The whole GR model is a composite actor contained by the top-level DE model. The GR model uses two inputs from the sampler -

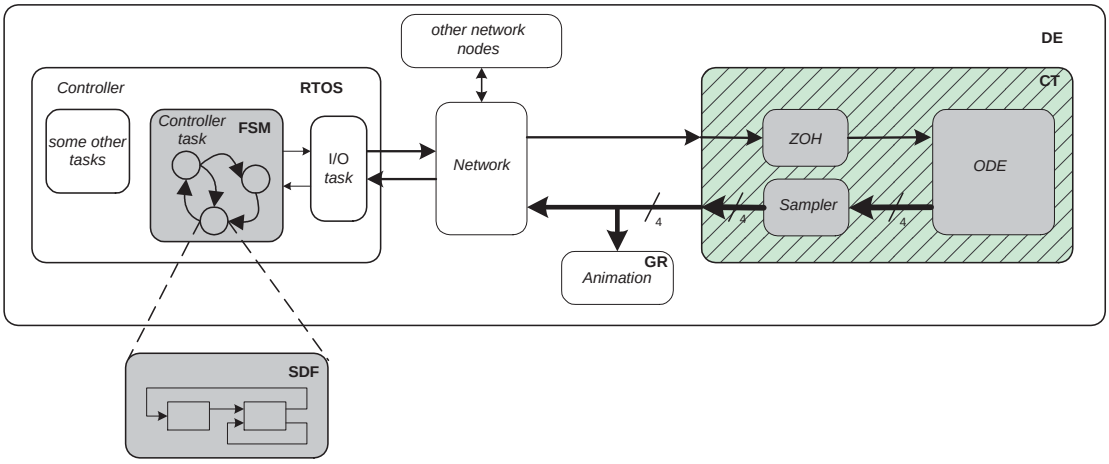

Fig. 5. A refined controller model, which also models execution times and communication latencies and allows us to study how they influence the closed loop performance.

the horizontal angle for the arm and the vertical angle for the pendulum. Since the GR model only represents object geometry, containment, and motion, the velocities are not used as inputs for the animation. It is possible that angular velocity values are needed to produce motion blur of the animation, but we did not implement it. Fig. 6 shows an image from our animated simulation of the pendulum.

5. CONCLUSION

We presented an approach to create models of heterogeneous control systems by following a discipline of hierarchical composition of various domains. A domain implements the rules of communication and control flow between model components in that domain. Domain interactions are well-defined, since they are based on a common framework notion used to express these models of computation. As a consequence, model components are reusable in almost arbitrary contexts.

We have demonstrated these concepts by taking a small control model and embedding its components in environments that better represent real-world influences on the actual system. We also showed how to use the same concepts to incorporate three-dimensional animation and interaction capabilities into the model, which significantly aid in understanding controller behavior and locating problems.

Further research is needed in identifying, defining, and implementing a more complete collection of domains pertinent to control system modeling. As domains form the semantical basis of the modeling framework, it is essential that they be orthogonal and that they interoperate in meaningful ways with a minimum of emergent behavior.

References

Christos G. Cassandras. *Discrete Event Systems: Modeling and Performance Analysis*. Richard D. Irwin Publ., 1993.

Johan Eker and Anton Cervin. A Matlab toolbox for real-time and control systems co-design. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, pages 320–327, Hong Kong, P.R. China, December 1999.

Chamberlain Fong. Discrete-Time Dataflow Models for Visual Simulation in Ptolemy II. Memo M01/9, UCB/ERL, University of California at Berkeley, 2001.

Alain Girault, Bilung Lee, and Edward A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 18(6):742–760, June 1999.

John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83 (5):773–801, May 1995.

Jie Liu. Continuous time and mixed-signal simulation in Ptolemy II. Memo M98/74, UCB/ERL, EECS UC Berkeley, CA 94720, July 1998.

Jie Liu and Edward A. Lee. Component-based hierarchical modeling of systems with continuous and discrete dynamics. In *2000 IEEE International Symposium on Computer-Aided Control System Design, Anchorage, Alaska, USA*, pages 95–100, September 2000.