REALISTIC SIMULATIONS OF EMBEDDED CONTROL SYSTEMS

Jie Liu, Johan Eker, Jörn W. Janneck, Edward A. Lee

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

Abstract: Current tools for design and implementation of embedded systems lack sufficient support for handling the different development phases. These phases usually include design of control laws, supervision logic, real-time task scheduling, modeling of communication, etc. The heterogeneity of modern embedded control systems puts high demands on design and simulation tools. Many control systems are hybrid in the sense that they consist of a set of subcontrollers and some switching logic. The individual subcontrollers are conveniently described using discrete equations, the switching logic may be expressed using state machines, the complete controller is implemented as a task in a real-time operating system, the controlled plant is modeled as a system of differential equations, and so on. The Ptolemy project addresses the design, simulation and implementation of heterogeneous hierarchical systems. The design principles are demonstrated in the Ptolemy II software which 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, such as those caused by real-time scheduling and network communication, are taken into consideration to achieve a more realistic simulation. *Copyright 2002 IFAC*

Keywords: simulators, hierarchical structures, real-time computers, software tools

1. INTRODUCTION

Designing an embedded control system is a complex and error prone task. Embedded systems in general consist of several subsystems and consequently the design problem is divided into a set of subproblems, e.g. deriving the actual control laws, allocating computing resources, dividing communication bandwidth, etc.

The development of embedded control systems usually consists of several distinct phases, and it is not uncommon that the work in different phases is executed by different people with specialized knowledge for the respective domain (*domain experts*). Typically, different tools are used in different phases, producing very different kinds of models for related aspects of the system, emphasizing one aspect of the system over others. Figure 1 shows an example of a classical development cycle with several stages and explicit handovers.

In the first phase, the control engineers develop control laws that must fulfill the given specifications. Control engineers typically make assumptions, like sampling rates and computational delays, based on preliminary knowledge of the plant, idealized sensor and actuator locations, and guesses about the implementation platform. The tool set for control engineers consists mostly of continuous-time and discrete-time simulators. At the end of the control design phase, there is a design review and the control algorithms are handed over to the implementation team.

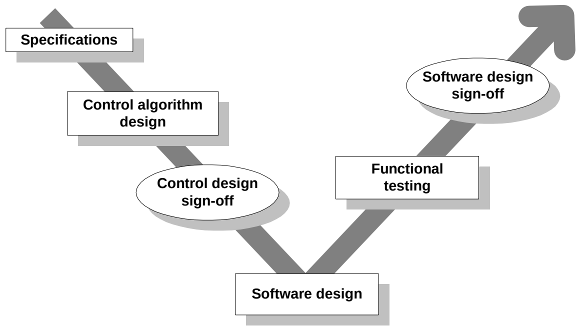

Fig. 1. The classical development cycle with several distinct phases. Each phase is reviewed and finalized before handed over to the next design team.

The implementation team then proceeds to choose a hardware platform and implement pieces of software that will execute the control law according to the specifications. They may immediately find that the sensors may not have the desired sampling rate as required by the control team, the controller may have to share resources with other control loops, and the delays may not be constants (as typically assumed by the controller designers) due to computation and communication jitter. All these problems are eventually solved by hacking the algorithms and tweaking priorities on the underlying RTOS, without knowing that some control loops may be more robust than others, and thus may tolerate more delay jitter.

The development relies on the testing phase following the design cycle. Errors found at this point usually indicate that some implementation decision violated a design assumption, but it is often very hard to localize either of them. This makes the design process time-consuming and fragile, since a slight change of the control specification may require a complete cycle of redesign. The reason is that the development was performed as if the different phases were orthogonal to each other, when in reality they are very much coupled.

This paper, in continuation of the work presented in (Eker et al., 2001), suggests that the iteration among the development phases can be performed more efficiently if all phases are supported by the same tool environment. This facilitates not only easy hand-over and integration of design artifacts, it also significantly improves component reuse and allows for a degree of concurrent engineering of different parts of the system.

Section 2 advocates a component-based design methodology that integrates heterogeneous modeling techniques usually encountered by embedded control system designers. The Ptolemy II environment, which uses hierarchical nesting of models to scope their heterogeneity, is presented as a candidate that implements a disciplined component-based design framework. A small example problem is introduced in Section 3 to show the gap between paper designs and the realistic implementations of an embedded controller. Section 4 shows how we migrate the embedded controller from abstract control laws to implementation by gradually adding design concerns and adjusting control laws accordingly. The model refinement is achieved within the Ptolemy II framework reusing model components designed in earlier phases.

2. COMPONENT-BASED DESIGN METHODOLOGIES

Many aspects of embedded control system may affect the final closed-loop control performance. One fundamental problem is how to decompose an embedded control system into more manageable and domain-specific subsystems, such that designers can effectively divide and conquer the problem. A component-based design methodology advocates an approach that decomposes the system into components with well-defined interfaces. Each of these components encapsulates certain functionality, such as computation and communication.

However, the kind of dynamics involved in the various kinds of models that occur in the development of an embedded control system can be very different, and so can the component interaction styles. For example, the dynamics of the plant is continuous, while the dynamics of the embedded controller is discrete. Within the controller, the dynamics of control laws, switching logic, real-time scheduling and communications are also different—synchronous or asynchronous, buffered or unbuffered, sequential or parallel, etc.

While the theories for each of the separate areas are relatively well understood and established, the integration of these dynamics brings significant complexity to the design problem, and the traditional approach that relies heavily on final testing and simulation does not scale well.

Hierarchical heterogeneity is a scalable way to manage the different kinds of dynamics. An aggregation of components with the same interaction style forms a composite component, which can in turn be integrated to another component, giving rise to a hierarchy of components. In the Ptolemy project (II et al., 2001), the different component interaction styles are characterized as formal *models of computation*. The Ptolemy II modeling and design environment implements many of these models of computation as *domains*, and supports hierarchical composition of different domains to realize heterogeneous designs. It 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.

The heterogeneity of modern embedded control systems puts high demands on design and simulation tools. The traditional design methodology of embedded control systems (modeling, control law derivation, simulation, and implementation) requires different skills and usually different tools for the task in the various phases, which are therefore often executed by different people. For example, the person deriving the control laws may not be the same as the one designing the software framework or scheduling the task allocation.

We believe that being able to execute the design of the different components and their integration into the final system within a common framework will significantly speed up design and allow to effectively reuse components developed at earlier stages. It will allow domain experts to communicate and collaborate more effectively, thereby alleviating a major impediment to design productivity.

These requirements make Ptolemy II an interesting candidate for modeling and designing embedded control system from control algorithms to implementation details. The extensive component libraries and hierarchical compositionality of heterogeneous models make Ptolemy II suitable for refining designs from conceptual models to implementation. 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.

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. Communication channels among actors are established by connecting ports. 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. The execution order, i.e. the control flow, of the actors in a composite is determined by a *director*. Each composite actor in a hierarchal model may have a different director, allowing a truly heterogeneous model. A model may be heterogeneous in the sense that there may be different policies governing the control flow (and also the communication) between actors at different levels of the hierarchy.

A domain defines the communication semantics and the execution order among actors. While the director regulates the flow of control between actors, the communication mechanism is implemented using *receivers*. To ensure a well-defined model of computation at each level of the 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.

A wide variety of domains has been implemented in Ptolemy II. The following is a subset of the Ptolemy II domains which occur in the subsequent examples. A more thorough discussion of models of computation is found in (Lee, 2000).

- Continuous-time (CT) domain models ordinary differential equations.
- Discrete-event (DE) domain models components that process a discrete sequence of timed events.
- Finite-state-machine (FSM) domain models modal operations and supervision logics.
- Real-Time Operation System (RTOS) domain models priority-based real-time scheduling of tasks that compete for shared resources.
- Synchronous-dataflow (SDF) domain models static structured dataflow style numerical computations.

In the example in section 3 the continuous process is modeled in CT, the switching logic of the hybrid controller in FSM, the algorithms of the subcontrollers in SDF and the behavior of the embedded software in RTOS.

Figure 2 shows a screenshot of an example Ptolemy II model. It is a simple two level hierarchical model. The fact that it is heterogeneous is reflected in the use of different directors at different levels in the hierarchy.

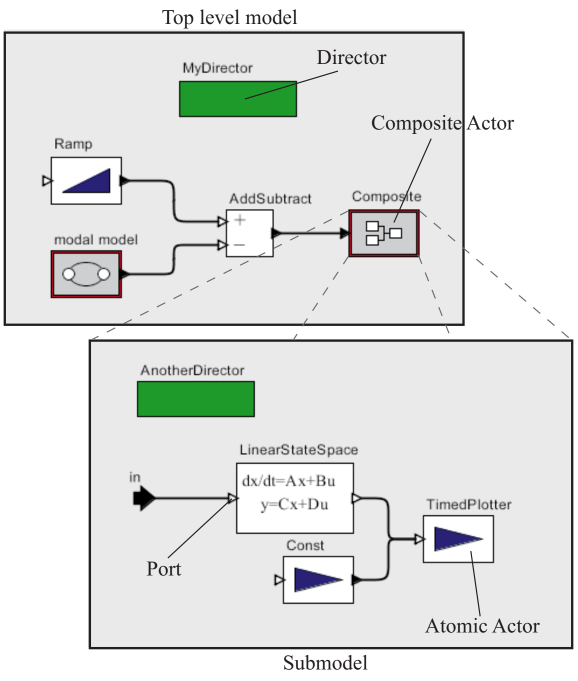

Fig. 2. A screenshot of a Ptolemy II model. The figures shows a two level hierarchical model. The directors of the different levels differ and provide a heterogeneous execution.

3. THE INVERTED PENDULUM CONTROLLER

The inverted pendulum is a classic control problem basically for two reasons: it is nonlinear and it is unstable. A picture of the Furuta pendulum is shown in Figure 3. The pendulum consists of two moving parts, the arm that moves in the horizontal plane and the pendulum that moves in the vertical plane. The goal of controller design is to swing-up and stabilize the pendulum.

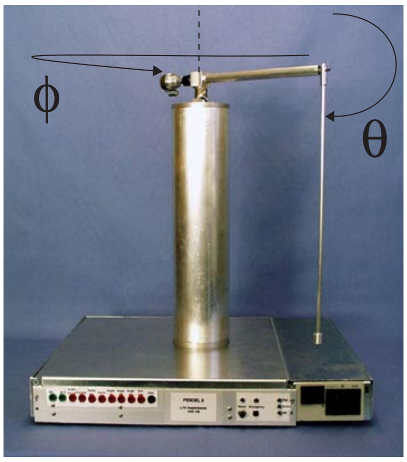

Fig. 3. The rotating inverted pendulum. The output signal from the process in the position of the arm φ and the position of the pendulum θ. The system is controlled by a torque on the arm.

It is straightforward to come up with a "paper design" of a hybrid controller that achieves the goal. Such a controller could consist 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 switches to the catch mode. The task of the catch mode is to reduce the speed of the arm before the third mode, stabilize, is entered. The mode switching logic can be described using a finite state machine, where each state corresponds to a control mode, and the individual controllers may simply be continuous-time functions.

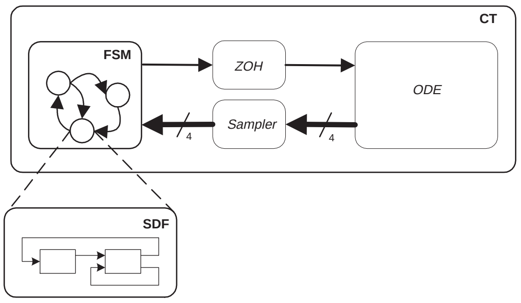

Fig. 4. A basic controller model. The controller is a discrete controller with three different modes, the plant is a continuous model, and they are connected via a zero-order hold (ZOH) and a sampler.

A step further from the continuous-time control law is to discretize the system and design discrete control laws. Discretization of nonlinear systems can be difficult, and simulation tools become useful to understand the control performances.

The above system is straightforward to model and simulate in many modern design tools. An outline of such a model is shown in Figure 4, where the modal controller is integrated with the plant dynamics, which is modeled as ordinary differential equations (ODEs). The output from a simulation of this model is shown in Figure 5. The pendulum angle θ is shown as a function of time. In the simulation the pendulum starts in the downward position ($\theta = \pi$) and swings up to the upward position ($\theta = 0$).

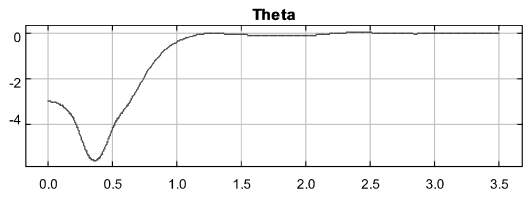

Fig. 5. The pendulum angle during a simulation of an ideal system. The y-axis shows the angle in radians and the x-axis is time is seconds. The simulation starts with the pendulum hanging down and reaching the upward equilibrium in less than 1.5 seconds.

However, the "paper designs" do 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.

These implementation-level characterizations may significantly affect the control performance. Gradually adding these concerns and migrating the controller from algorithms to implementation within one framework can bridge the gap between the algorithmic team and the implementation team. Innovations on control laws, communication protocols, hardware platforms, and scheduling algorithms can be tightly integrated. In this process of migrating to the implementation, components developed at the early stage of the project will be reused.

4. BRIDGING TO REALITY

In this section, we gradually add realistic concerns to the pendulum control system. These concerns are shared computing resources and communication networks.

4.1 *Real-Time Scheduling*

We first add the consideration of real-time scheduling policies by modeling the behavior of the controller on top of a real-time operating system (RTOS). This is done in Ptolemy II by embedding 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.

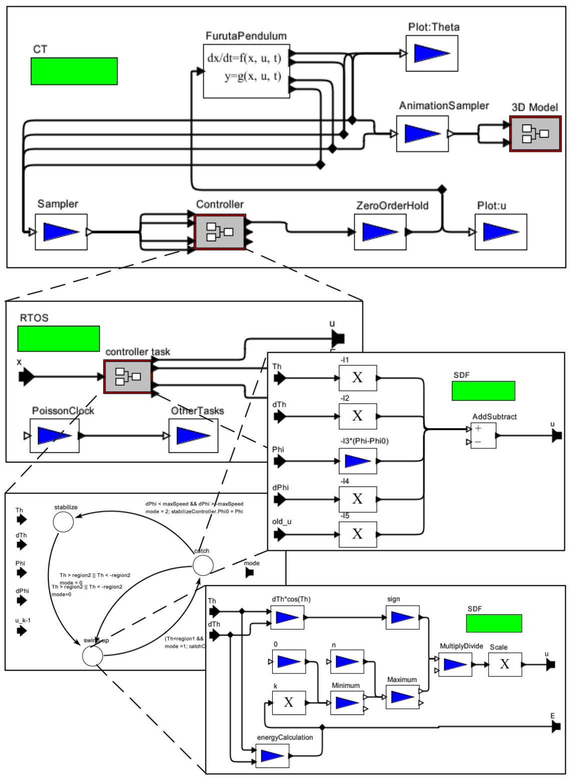

Fig. 6. A refined controller model, which also models execution times and communication latencies and allows us to study how they influence the closed loop performance.

Figure 6 shows the modeling of this step in Ptolemy II. The top-level is a continuous-time domain with one (composite) actor modeling the pendulum dynamics and one composite actor modeling the controller. Inside the controller composite, we use the RTOS domain, where the controller (reused from algorithms design) is treated as one software task. A Poisson process triggers another actor, which models other software tasks. This task can be an abstract model of the I/O part of the controller, 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.

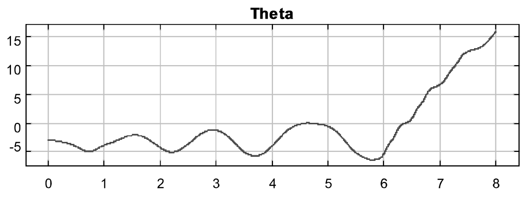

Fig. 7. The pendulum angle during a simulation of the system running under the RTOS model. Due to the latencies introduced, in this more realistic simulation model, the controller has problems catching and stabilizing the pendulum.

The RTOS domain manages the tasks by their *priorities* and *execution time*. The flexible software architecture of this domain allows any real-time scheduling algorithms to be hooked to it to provide the priority values at run-time. The execution time needs to be added manually. Potentially, these numbers can be obtained from implementation platform architecture and worst case execution time analysis tools. Notice that the composite actor we built in the basic model only specifies the computational part of the controller, which make it reusable in the RTOS domain.

Figure 7 shows the result of simulating the original controller as it would be running under the RTOS, demonstrating the effects of the delays due to its actual execution time and its interaction with other tasks. We find that due to these delays the controller is no longer able to swing-up and stabilize the pendulum. By looking at the simulated execution schedule we are able to determine the type of delay that occurs and compensate for it using standard text book solution. When the control law is modified the simulation is run again and the output is presented in Figure 8. The controller is again able to control the pendulum, but it now needs almost five seconds to bring the pendulum up and this decrease in performance is due to the delays.

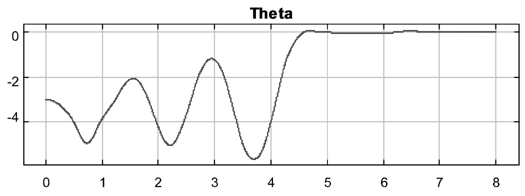

Fig. 8. The pendulum angle during a simulation of the system using a modified controller simulated under the RTOS model.

4.2 *Distributed Controller*

The next step is to include a model of the network communication. This is done by using a hierarchy of discrete-event domains. The `Controller` actor at the top level of Figure 7 is replaced by the composite actor shown in Figure 9. The composite actor contains five actors—one network and four hosts—in the DE domain. The `RTOS/Sensor` packages sensor readings into network packets. The packet is disassembled by the `RTOS/Controller` and the data are fed into the controller in Figure 7. Similarly, the outputs of the controller are packaged into network packets and sent to the actuator host. A forth host is used to model other nodes on the network. The network composite actor is internally implemented by transmitters, receivers, and a model of the communication media. Figure 9 shows an CSMA/CD style media access protocol, as seen in Ethernet. The transmitters try to send packets and listen to possible collisions. Only when no collision occurs during the entire sending period will the receivers receive the packets. The receiver filters the packets and output only the ones that are directed to the corresponding actor.

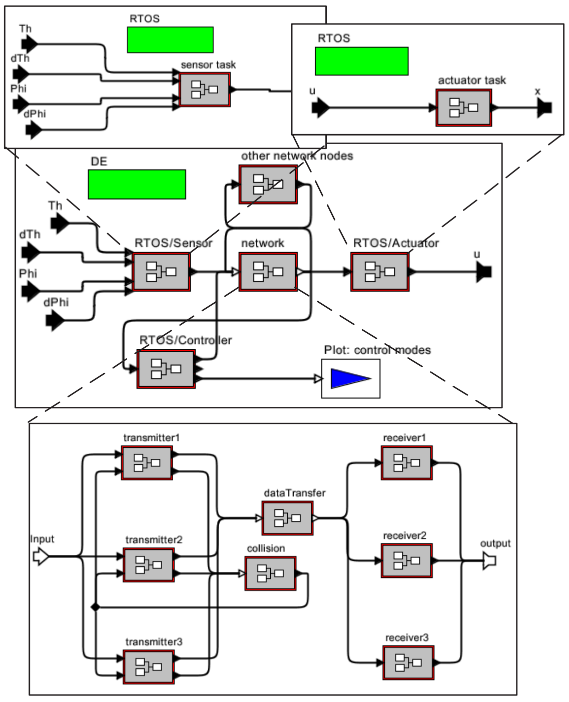

Fig. 9. Modeling the inverted pendulum control system in Ptolemy II.

Similarly to the previous section, where we dealt with delays due to execution time and preemption, we now must do the same to handle delays introduced due to communication. This will again require a redesign of the control laws and the switching conditions.

4.3 *Future work: Hardware-In-The-Loop Simulation*

Current efforts are in implementing hardware-in-the-loop simulation that brings the model even closer to reality. A real controller and a real network, implemented on embedded hardware and software, replace the `Controller` block at the top level of Figure 8. The dynamics of the pendulum, together with sensors and actuators, are still simulated in Ptolemy II. The component-based architecture defines a clear boundary for such integration. Furthermore, the reactivity of the controller under the RTOS makes the Ptolemy model the master of the simulation and controls the progression of time.

5. CONCLUSION

This paper suggests the use of hierarchical heterogeneity as a design methodology for developing embedded control systems, and demonstrates its application using the Ptolemy II modeling environment. This methodology supports the integration of heterogeneous models into a well-behaved and understandable system. This facilitates the use of one design environment throughout the development of the control system, and aides the collaboration among the various groups of domain experts involved in a project.

Traditionally, the control design is usually made based on assumptions about the execution, i.e. fixed, known sampling rates and delays. If these assumptions do not hold in the actual implementation the control may become unstable or perform badly. By simulating not only the controller and the controlled process but also the software and network behavior, it is possible to detect problems early in the development cycle and correct them. Hence, the simulation software must allow submodels of very different nature to be composed and simulated. In this paper the Ptolemy II package has been use to construct a highly complex model which captures many important aspects of the final embedded control system.

References

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.

Johan Eker, Chamberlain Fong, Jörn W. Janneck, and Jie Liu. Design and simulation of heterogeneous control systems using ptolemy ii. In *IFAC Conference on New Technologies for Computer Control*, Hong-Kong, China, November 2001. IFAC.

John Davis II, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, Jeff Tsay, Brian Vogel, and Yuhong Xiong. Heterogeneous concurrent modeling and design in java. Technical Memorandum UCB/ERL M01/12, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley California, USA, March 2001.

Edward A. Lee. What's ahead for embedded software? *IEEE Computer*, 33(7):18–26, September 2000.