Submited to *IEEE Tran. on Control System Technology,* special issue on Computer Automated Multi-Paradigm Modeling. March 2002.

Actor-Oriented Control System Design

Jie Liu
Palo Alto Research Center
3333 Coyote Hill Rd., Palo Alto, CA 94304
liuj@parc.com

Johan Eker
Department of Automatic Control
Lund University, Sweden
johane@control.lth.se

Xiaojun Liu, John Reekie, Edward A. Lee
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720
{liuxj, johnr, eal}@eecs.berkeley.edu

Abstract: Complex control systems are heterogeneous, in the sense of discrete computer-based controllers interacting with continuous physical plants, regular data sampling interleaving with usually irregular communication and user interaction, and multilayer and multimode control laws. This heterogeneity imposes great challenges for control system design technologies in terms of end-to-end control performance modeling and simulation, traceable refinements from algorithms to software/hardware implementation, and component reuse. This paper presents an actor-oriented design methodology that tames these issues by separating the data-centric computational components (a.k.a. *actors*) and the control-flow-centric scheduling and activation mechanisms (a.k.a. *frameworks*). The underlying principle of frameworks is to use formal *models of computation* to manage the interactions among actors. Semantically different frameworks can be composed hierarchically to manage heterogeneity, improve understandability, and achieve actor and framework reusability. This methodology is implemented through the Ptolemy II software environment. As an example, the methodology and the Ptolemy II software have been applied to the design of a pendulum inversion and stabilization system.

1. Introduction

Embedded control system design is a complex and error prone task, not only because the algorithms implemented in these systems contain significant amount of domain-specific expertise, but also because these systems are typically heterogeneous. Control systems constantly interact with the physical world. The nature of the problems requires that the embedded computer systems be reactive, real-time, non-terminating, and collaborative. Modern computer-based control systems usually consist of discrete controllers interacting with continuous plants, regular sampled-data computation interleaving with irregular communication and user interaction, and multilayered multimode tasks with different time scale and latency requirements. These complexities challenge the design of control systems in many aspects, such as closed-loop control performance analysis, design refinement and realization, modular testing, and component reuse.

An example of a classical control system design process is shown in Figure 1. The process consists of several distinct phases with explicit hand-overs. A system is conceptualized in some informal specification and handed over to control engineers for control law design. Control engineers obtain (usually simplified) plant models by system identification or physical modeling. Plant models are typically continuous, and

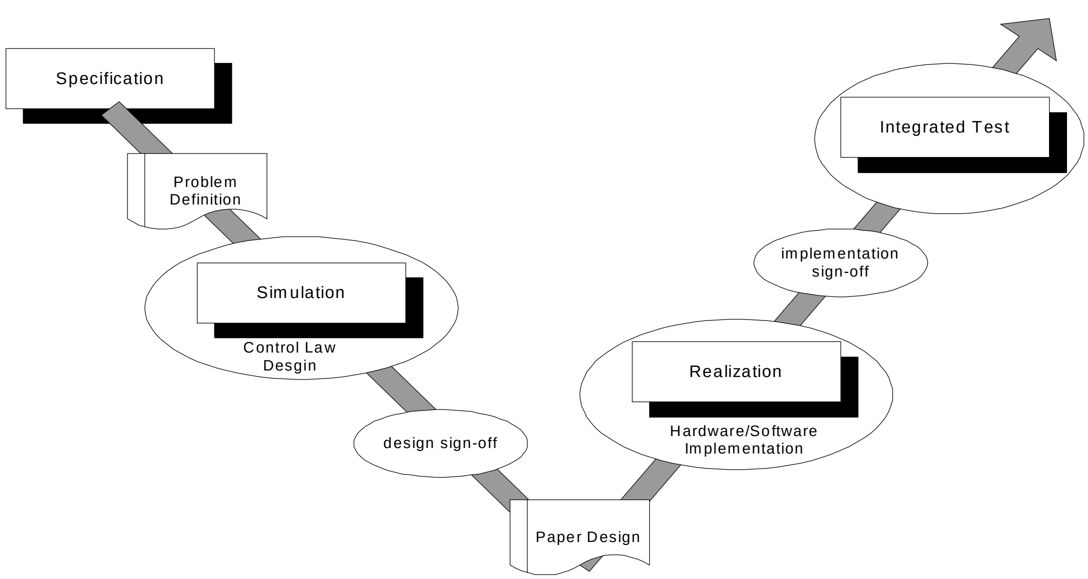


Figure 1. The traditional development cycle with distinct phases

continuos control laws are derived to fulfill performance requirements like stability, responsiveness, and robustness. The control laws are typical validated by mathematical proofs and simulation in the continuous-time domain. If this is not the end of the control law design phase, control engineers may discretize the control law with ideal assumptions, such as sampling rate, computational delays, sensor and actuator locations, communication networks, and perhaps the implementation platform. At the end of the control design phase, there is a design review and the control algorithms, in the form of formulae on paper, are handed over to the implementation team.

The implementation team chooses a hardware platform and implements pieces of software that will execute the control law according to the specifications. They may immediately find that the sensors do not have the desired sampling rate as required by the control team, that the controller may have to share resources with other tasks, or that 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 adjusting the algorithm parameters and tweaking priorities on the underlying RTOS in an *ad hoc* way, without formal understanding of closed-loop control performance and their relative robustness.

The development relies on the integrated testing phase following the design cycle. Errors found at this point usually indicate that some implementation decisions violated design assumptions, 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.

One problem behind this process is the distinct expertise required in various stages and the different modeling paradigms used in each stage and each subsystem. Not too many control engineers are trained with enough software engineering knowledge to explore complicated software design, and vice versa. The consequences are gaps in system modeling and jumps in the design process. The development was performed as if the different phases were orthogonal to each other, when in reality they are very much coupled.

A much more integrated design process is presented in Figure 2. Here the control design team and the system design team both work on the same model. Changes in the implementation structure are reflected in the control performance and, similarly, modifications to the control laws are reflected in the software architecture. Closing the loop between control engineers and software engineers is not easy. The modeling technologies developed for each individual field are highly specialized and the domain engineers have their specific ways of thinking. We not only need system theories (like hybrid systems) that integrate more than one kind of dynamics, but also need design methodologies and modeling tools that help designers decompose a system into domain-specific subsystems and recompose these components into a coherent and realistic system-level model. In particular, the use of abstraction may bring software architecture decisions close to control engineers.

This paper presents an actor-oriented design methodology for complex control systems. Actor orientation gives the flexibility of exploring the design space with refinement, composition, implementation, realization, and perturbation, with well-defined scope. In addition, we advocate the use of formal models of computation to guide the interaction styles among actors, and build hierarchically composable frameworks to enhance the modeling capability of a design environment. This methodology is being implemented through the Ptolemy II modeling and design environment [8].

The remainder of the paper is organized as follows. Section 2 describes the actor-oriented design methodology, which advocates the decomposition of systems into interacting components (actors) and recompose the components with well-defined models of computation. Section 3 discusses model-of-com-

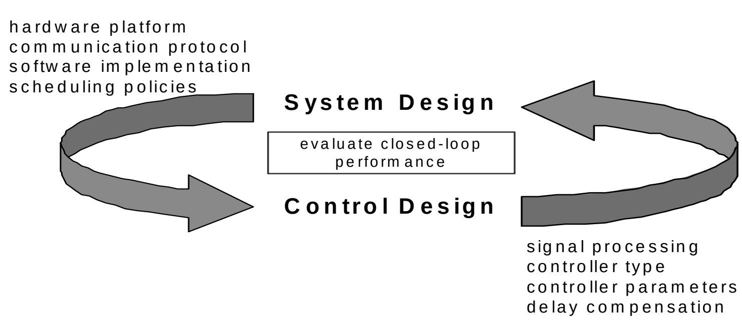


Figure 2. In a more integrated design flow, control designs and system implementation are tightly coupled, so that design decisions can be quickly evaluated and fed back between phases.

putation frameworks that gives rigorous semantics for component interaction, and the hierarchical composition of heterogeneous frameworks. In section 4, we present the Ptolemy II design environment that implements an actor-oriented design methodology. An example of designing an inverted pendulum control system in Ptolemy II is given in section 5.

2. Actor-Oriented Design Methodology

Many aspects of a control system may affect the final closed-loop control performance. One fundamental problem is how to decompose a control system into more manageable and domain-specific subsystems, such that designers can effectively divide and conquer the problem. Component-based design methodologies advocate approaches that decompose a system into components with well-defined interfaces. Each of these components encapsulates certain functionality, such as computation and communication.

There are many examples of component-based design methodologies, which provide different ways of viewing components, such as object orientation, middleware orientation, and actor orientation [29]. Object-oriented (OO) design manages complexity in the system through object abstraction, class hierarchies, and method call interfaces. This methodology has been adapted to design embedded and real-time software, emphasizing the use of UML [15] to formally specify systems. Object-oriented software environments, such as Rational Rose [7], GME [17], and DOME [31], have been applied in control system designs.

Noticing that some objects usually work together to provide a logical piece of functionality, middleware-oriented design advocates the encapsulation of one or more objects into conceptual services, and composing services into a system. The power of middleware services is more significant in distributed systems, since the notion of communication may be much cleaner than remote procedure calls in general OO frameworks. Thus, they appear more often in large-scale applications, which leverage distributed object infrastructures, such as CORBA [30], DCOM [27], and JINI [16]. The open control platform (OCP) [25] developed at Boeing is an example of middleware-oriented design for real-time control systems.

Despite their logical differences, the basic structure of object-oriented and middleware-oriented systems are objects that are related to each other by references. Their primary interaction interface is method calls. A method call directly transfers the flow of control from one object to another. Important system characteristics, such as concurrency and communication, are hidden behind the method call interface. As a consequence, both object-oriented and middleware-oriented design methodologies emphasize how to decompose a system into components, but the composition of components is left to designers.

2.1 Actors

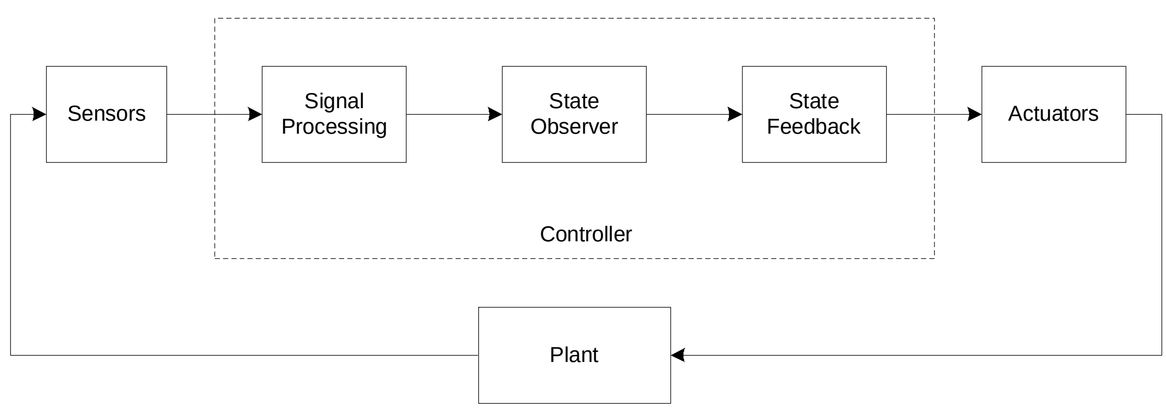
Actor-oriented designs decompose a system from the view point of “actions” in the system. An *actor* is an encapsulation of parameterized actions performed on input data and produce output data. Actions may be stateless or stateful depending on whether it has internal state. Input and output data are communicated through well-defined *ports*. Ports and parameters are the interface of an actor. A port, unlike methods in object-oriented design, does not have call-return semantics. Essentially, an actor defines *local* activities without implicit referencing to other actors.

There are many examples of actor-oriented design frameworks, including Simulink from MathWorks, LabVIEW from National Instruments, SPW from Cadence, Cocentric studio from Synopsys, and ROOM (Real-time Object-Oriented Modeling [26]) from Rational Software. In the academic community, active objects and actors [1] [2], port-based objects [28], hybrid I/O automata [23], Moses [11], Polis [4], Ptolemy and Ptolemy II [8] all emphasize actor orientation.

An important issue to be answered by actor-oriented frameworks is the interaction styles among actors. This also differentiates many actor-oriented modeling paradigms. For example, ROOM and Agha’s actor suggest that actors be “active;” that is, each of them has its own thread of control. Some others, like the ones in Simulink, LabVIEW, and SPW, however, do not have active actors. Instead, a central scheduler determines the flow of control among the actors based on the underlying semantic model. We capture the interaction styles of actors by the notion of *model of computation* (MoC). A MoC defines the communication semantics among ports and the flow of control among actors. An implementation of a MoC is a *framework* that the actors reside in. Frameworks and actors together define a system. Most actor-oriented modeling environments have a unified MoC and implement a single framework. For example, Simulink is a continuous-time/mixed-signal framework, while SPW is a dataflow framework. But for complex control systems, a single MoC is usually not enough.

Figure 3 shows an actor decomposition of a simple control system into sensors, a signal processing unit, a state observer, a state-feedback controller, actuators, and a plant. Thinking in terms of actors and frameworks maps well to the actual partition of a system, and it helps to identify different concurrency and communication issues. For example, in Figure 3, the plant operates concurrently with the controller in the physical world, but, within the controller, the three actors implemented by embedded software may operate sequentially because of data dependencies. The interaction between the plant, sensors, actuators, and the controller has a continuous-time/mixed-signal style, while within the controller, a dataflow model may be more applicable.

The MoC that guides the interaction of actors reflects the “dynamics” among subsystems, which could be diverse even in our simple control system –– the dynamics of the plant is continuous, while the dynamics of the controller is discrete. In more complicated cases, even within the controller, the dynamics of control laws, switching logic, real-time scheduling, and communication networks are also different ––



synchronous or asynchronous, buffered or unbuffered, sequential or parallel, preemptive or nonpreemptive, and so on. 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. If a design methodology only supports a single MoC, then correctness of a design has to rely on final integrated testing, which typically leads to long design cycle and high cost.

2.2 Hierarchical Heterogeneity

A powerful concept in actor-oriented design is the use of *hierarchy*, which suggests that a network of actors can be viewed as a single actor “from a distance.” Using hierarchy, one can effectively divide a complex model into a tree of nested submodels, which are composed at each level to form a network of interacting components. Hierarchy is a particular kind of abstraction, and hides the detail of a subsystem from the rest of the system, so that interaction and modification in that subsystem are restrained within that level.

Hierarchies can be used in unified models to manage syntactic complexity, as seen in Simulink. A more effective use of hierarchy is to mange heterogeneity in models of computation, an approach called *hierarchical heterogeneity* [10]. This approach constrains each level of interconnected actors to be locally homogeneous, while allowing different models of computation to be specified at different levels in the hierarchy. A well-defined model of computation at the same level improves the understandability of the system, and allows certain parts of the system to be correct by construction, because of the formal properties obtained by that specific MoC.

For example, Figure 4 shows a hierarchical model for the control system illustrated in Figure 3. The top level is a continuous model of the plant interacting with a *continuous* view of the controller, and within the controller, a dataflow model specifies the interaction among the three software components. Hierarchically composing heterogeneous models is not trivial. The interactions at the boundary of different models need to be carefully examined. We further discuss this in section 3.2.

2.3 Actor-Oriented Design Process

Using actors and frameworks enables a finer-grained view of the design process necessary to implement an integrated design methodology (Figure 2). Each step in the design process can be as simple as an operation on a single actor, such as refining it into a more detailed description, or replacing a simulation actor with a hardware realization. The operations we have identified as being sufficient to support this design process are:

Refinement:

Refinement is the process of decomposing the specification of an actor into more detailed specifications. A specification may take the form of one or more interfaces that capture specific sets of properties about an actor, including its functionality, interaction styles, timing behavior, resource requirement, and so on. Refinement typically reflects a top-down design philosophy. A system is first specified at a high level of abstraction without considering many implementation details. Then, more considerations are added to break down a coarse-grained specification into more implementable specifications. For example, Figure 4 may be seen as a step that the abstract controller is refined into a signal-processing unit, an observer, and a state feedback.

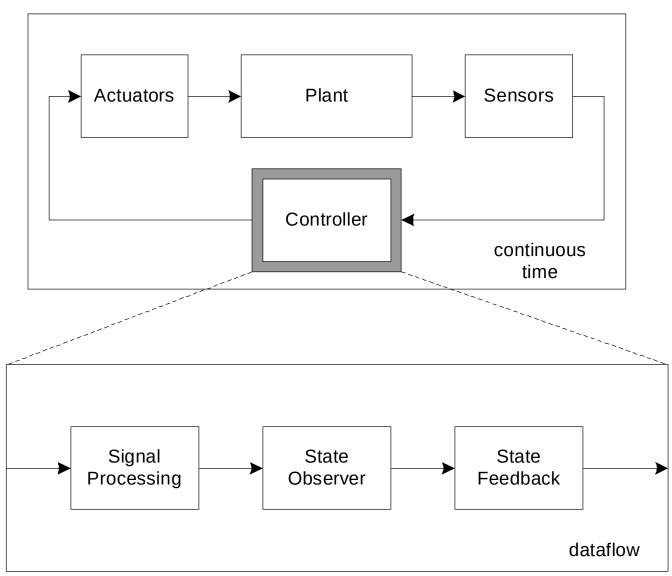


Figure 4. A hierarchical view of the control system in Figure 3.

Implementation:

Implementation is the process of replacing an actor's specification with a component that models the appropriate behavior. For example, each interface at some level of the system description is implemented by a single executable actor.

Composition:

Composition is the process of composing implementations into a composite (executable) component. Obviously, the composition is guided by a model of computation. Composition usually reflects a bottom-up, divide-and-concur philosophy in a design. For example, if a controller for situation A and another controller for situation B have been separately designed, then we may compose the controllers under a finite state machine model to obtain a modal controller for both situations. Interface theories [9] indicate that if each component implements an interface, then the composition of components can be formally shown to implement the more abstract interface.

Realization:

Realization is the process to replace parts of the model in a simulated system with physical objects. The difference between realization and implementation is that the object that realizes a subsystem is not part of our design. It is usually a given, such as the physical plant. Sometimes, the execution of a system that involves both implemented and realized actors is called *hardware-in-the-loop* simulation.

Perturbation:

Perturbation is the process of changing parameters in a design without changing the structure of the model.

Using actor orientation and hierarchical heterogeneous models, a design process may look like the one shown in Figure 5, comparing to Figure 1. In Figure 5, the three stages of design, specification, simulation, and realization, merge together. There may be system models, visualized as vertices in the design

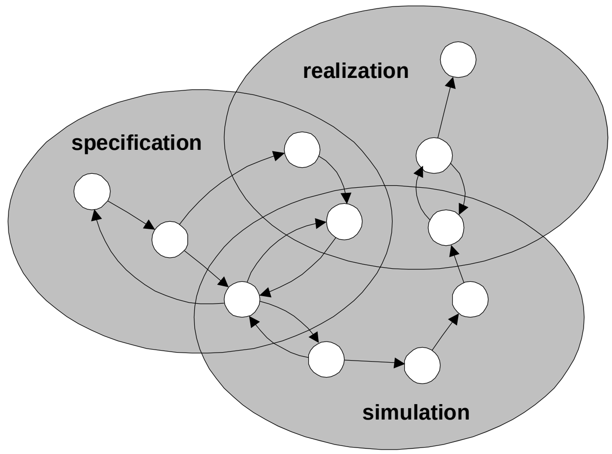


Figure 5. A seamless design process.

space, that are part specification, part simulation, and part realization. (Obviously, the specification parts won't execute, but they may be stubs that keep the model complete, or assertions that validate execution results.) We believe that being able to capture all steps of a design process 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.

3. Model of Computation Frameworks

A diverse set of models of computation has been proved to be useful in control system design practice. We discuss a subset of them in terms of actor-oriented formulation, and give a hierarchical composition model through the notions of *precise reaction* and *responsible frameworks*.

3.1 Frameworks for Control System Design

3.1.1 Continuous time

Continuous time (CT) models, in particular ordinary differential equations (ODEs), are widely used in control system design for modeling physical dynamics and continuous control laws. Using a special actor — the *integrator* — an ODE can be modeled as a feedback. For example, Figure 6 implements the following ODE: $\dot{x} = 2x + \sin(t) - 1$. Each connection in this model carries a continuous-time signal. And the actors denote the relations among these signals.

Interacting with other models requires that a CT framework be extended to allow the handling of discrete events. Such a model may be more appropriately called a *mixed-signal* model. Event generators, such as periodic samplers, triggered samplers, and level-crossing detectors, are actors that can convert continuous-time signals to discrete events. Waveform generators, such as a zero-order hold, are actors that convert discrete events to continuous-time signals.

The execution of a CT model involves the computation of a numerical solution to the ODEs at a discrete set of time points. In order to support the detection of discrete events and the interaction with discrete models of computation, the time progression and the execution order of a CT model must be carefully controlled [22].

3.1.2 Discrete event

In a discrete event model, actors share a global notion of time and communicate through events that are 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 widely used in modeling hardware and software timing properties, communication networks, and queuing systems.

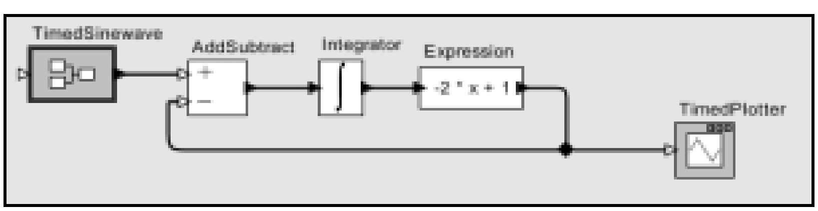


Figure 6. An actor-oriented differential equation model.

3.1.3 Dataflow models

In dataflow models [18], connections represent data streams, and actors compute their output data streams from input streams. In such models, the order of execution of the actors are only constrained by the data dependency among them. This makes dataflow models amenable to optimized execution, for example to minimize buffer sizes, or to achieve a higher degree of parallelism. Dataflow models are very useful in designing signal processing algorithms and sampled control laws.

There are many variants of dataflow models, of which synchronous dataflow (SDF) [19] is a particularly restricted special case. In SDF, when an actor executes, it consumes a fixed number of data tokens from each input port, and produces a fixed number of tokens to each output port. For a consistent SDF model, a static schedule can be computed, such that the actors always have sufficient data before execution. For algorithms with a fixed structure, SDF is very efficient and predictable.

3.1.4 Timed Multitasking

The timed multitasking (TM) model [21] allows designers to explore priority-based scheduling policies such as those found in a real-time operating system (RTOS) and their effects on real-time software. In this model, actors are software tasks with priorities. The framework of a TM model 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.

3.1.5 Synchronous/Reactive

In the synchronous/reactive (SR) model of computation [24], the connections among actors represent signals whose values are aligned with global clock ticks. Thus, they are discrete signals, but need not have a value at every clock tick. The actors represent relations between input and output values at each tick, and are usually partial functions with certain technical restrictions to ensure determinacy. Examples of languages that use the SR model of computation include Esterel [5] and Signal [4]. SR models are excellent for discrete control applications with multiple, tightly-coupled, and concurrent tasks. Because of the tight synchronization, safety-critical real-time applications are a good match.

3.1.6 Finite state machines

A finite state machine (FSM) has a set of states and transitions among states. An FSM reacts to input by taking a transition from its current state. (The transition may be an implicit transition back to the current state.) Outputs may be produced by actions associated with the transition. FSMs are very intuitive models of sequential control logic and the discrete evolution of physical processes. FSM models are amenable to in-depth formal analysis and verification [14]. Applications of pure FSM models include datapath controllers in microprocessors, communication protocols, and control logics.

In a hierarchical heterogeneous modeling environment, finite state machines provide two modeling mechanisms. One allows designers to create actors whose behaviors are specified by FSMs. We can think of this as a graphical scripting language for writing new actors. The other one applies to modal models, which are hierarchical composition of FSMs with other models of computation [13]. For example, hybrid systems are hierarchical composition of FSM and CT models.

3.2 Composing Frameworks

In order to facilitate hierarchical composition, a MoC must be compositional in the sense that it not only aggregates a network of components, it also turns that network itself into a component which in turn may be aggregated with other models, under a possibly different MoC. We describe the notions of precise reactions and responsible frameworks as means of studying compositionality of model [20].

A key contribution of the notion of actors is the localization of computation within the actors. An actor references nothing but its ports and the framework. An actor may or may not have its own thread of control. We call an actor *reactive* if its computation and communication need to be triggered by the framework, and in response, it finishes a *finite* amount of computation then waits for the next trigger. Unlike synchronous/reactive models, we do not necessarily assume a reaction to be *synchronous*. A reaction, in general, takes time, costs resources, and requires data inputs. When developing actors, designers usually have logical notions on what a reaction consists of, and what state the actor should be in at the end of the reaction. This state, in which the actor has well-defined values in its state variables and has released all resources it occupied, is called a *quiescent state* of the actor.

A reaction, in general, may not finish at a quiescent state. For example, in an unmanaged multithreading model, an actor may do blocking read on its input ports. If there are not enough data provided at the ports, the actor may be blocked at a non-quiescent state. This may be hazardous for model composition. For example, if a mode switching is performed at a higher level, this actor is left at an inconsistent state. For this reason, we introduce the concept of a *precise reaction*. A precise reaction is a reaction that once triggered, always finishes at a quiescent state of the actor. Figure 7 shows a general structure of a precise reaction, where each circle represents an operation within the reaction and the arrows represent the dependencies among operations. Notice that the operations may not be sequential. This is especially the case when an actor is internally constructed using a concurrent model of computation. Precise reaction indicates that if an (internal) operation of a reaction depends on other external operations (possibly in other actors), then the trigger of this reaction should only happen after those external operations have finished. Such a trigger is called a *responsible trigger* for this reaction. Precise reactions, although closely related to non-preemptable atomic reaction, do allow concurrency within a framework.

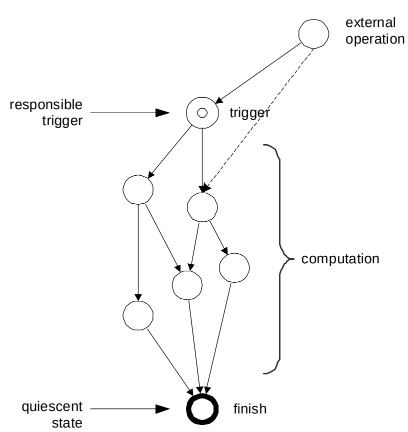


Figure 7. A general structure of a precise reaction.

A *responsible framework* is a framework that only produces responsible triggers. As a consequence, all reactions within a responsible framework are precise reactions. The advantage of a framework being responsible is that the quiescent state of all actors can be achieved simply by monitoring the triggers. If the framework stops sending triggers, all actors will eventually come to their quiescent states. This is in particular valuable in hierarchical heterogeneous modeling, so that a responsible framework with reactive actors may have a well-defined reaction, consisting of precise reactions of each individual actors. Thus, the composition can be treated as an *opaque* actor from a higher level.

Systems that engage the real world typically inherit physical properties from the real world, in particular, the notion of *time*. Many models discussed in section 3.1 have time as the driving factor for their executions. In fact, having the notion of time helps us build responsible frameworks. Timed models typically imply that a system has a well-defined state at a set of time points. For instance, for CT models, this set is the entire the real line (possibly with a starting time); for DE models, this set is the subset of the real line when there is an event happening; for discrete-time models, this set is the multiples of the sampling period; and for SR models, this set is the set of ticks. These well-defined (i.e. quiescent) states and the total ordering of time points reduce the responsible framework problem into the management of time progression among heterogeneous models.

4. Ptolemy II Design Environment

The Ptolemy II modeling and design environment implements an actor-oriented design methodology. Ptolemy II supports actor-oriented system construction and execution through hierarchical heterogeneity. In terms of the design process discussed in section 2.3, only the composition and some implementation operations are presently implemented. This is also the case with most other actor-oriented tools. We are working on adding support for the other types of transition to Ptolemy II, in particular, hardware-in-the-loop execution and interface refinements.

The basic building blocks in a Ptolemy II model are *atomic actors*. Atomic actors encapsulate basic computation, from as simple as an AND gate to more complex ones like an FFT. Through composition, actors that perform even more complex functions can be built. A composition of actors are guided by a director, which represents a model of computation framework and is visualized as an annotation box in the Ptolemy II graph editor. A model is a hierarchical composition of actors, as shown in Figure 8. The atomic actors, such as A1 and B1, only appear at the bottom of the hierarchy. Actors that contain other actors, such as the A2, are *composite*. A composite actor can be contained by another composite actor, so the hierarchy can be arbitrarily nested.

A director controls the execution order of the actors in a composite, and mediates their communication. In Figure 8, Director1 may choose to execute A1, A2, and A3 sequentially. Whenever A2 is executed, Director2 takes over and executes B1~B3 according to the model of computation it implements. A responsible Director2 gurantees that the flow of control will be returned to Director1, and A2 is at a quescent state when its local execution returns. A director uses receivers to mediate actor communication. As shown in figure 9, one receiver is created for each communication channel; it is situated at the input ports, although this makes little difference. When the producer actor sends a piece of data, called a *token* in Ptolemy II, to the output port, the receiver decides how the transaction is completed. It may put the token into a FIFO

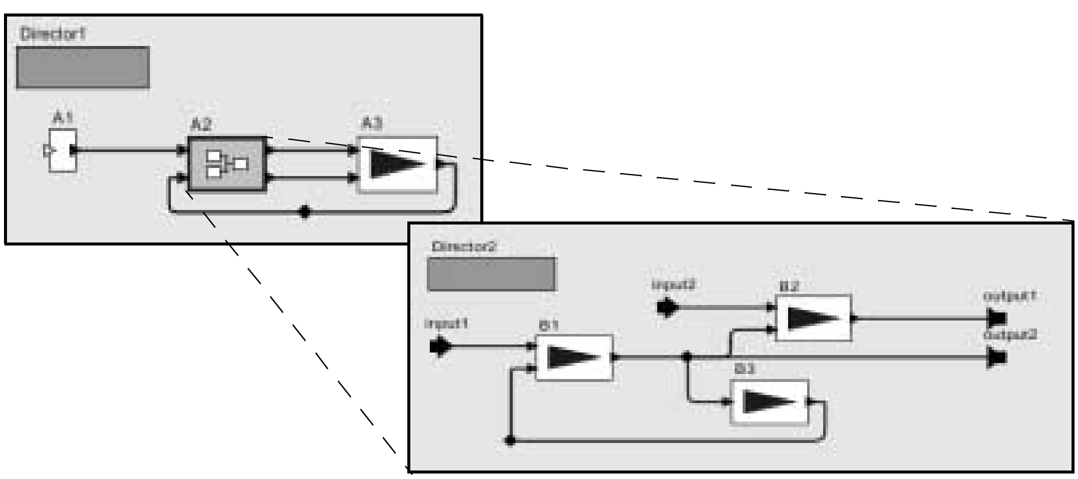


Figure 8. A hierarchical model in Ptolemy II.

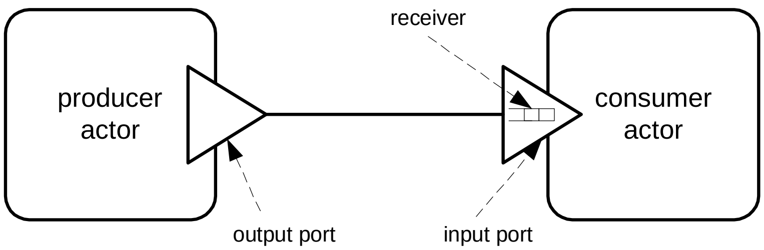


FIGURE 9. A receiver is used to mediate communication between actors.

buffer, from which the consumer actor will get data. It may tag the token as an event, and put the event in a global event queue. The token will be made available to the consumer when time comes for the consumer to process the event. Or it may stall the producer to wait for the consumer to be ready.

By choosing an ordering strategy and a communication mechanism, a director implements a model of computation. Each model of computation, including those described in section 3.1, is called a *domain* in Ptolemy II. Within a composite actor, the actors under the immediate control of a director interact homogeneously. Properties of the director's model of computation can be used to reason about the interaction. A heterogeneous system is modeled by using multiple directors in different places in the hierarchy. The directors are carefully designed so that they implement responsible frameworks and provide a polymorphic execution interface to the director one level up in the hierarchy. This ensures that the model of computation at each level in the hierarchy is respected.

5. A Design Example

The inverted pendulum is a classic control problem basically for two reasons: it is nonlinear and unstable. A picture of the Furuta pendulum is shown in Figure 10. 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 this controller design is to swing up and stabilize the pendulum.

Using an actor-oriented design methodology, we may go through the design process in the following steps:

1. continuous-time pendulum modeling;

2. continuous-time controllers for stabilization;
3. discrete-time controllers for stabilization with zero execution delay;
4. discrete-time modal controllers with three control modes –– swing-up, catch, and stabilize;
5. discrete controllers with RTOS scheduling;
6. discrete controller with network integrated sensors and actuators;
7. hardware-in-the-loop simulation with embedded implementation;
8. deployed realization.

Note that steps 1 to 6 are mostly implementation and composition processes, adding new concerns and components to the system model. Step 7 and 8 realize part or all systems physically. While the modeling and design within Ptolemy II has been achieved, steps 7 and 8 are ongoing work.

The first step is to create the dynamic model for the inverted pendulum, as developed in [12]:

$$
\begin{aligned}
\dot{x}_1 &= x_2 \\
\dot{x}_2 &= \frac{\alpha\beta + \alpha^2 Sin^3(x_1)Cos(x_1)x_4^2 - \gamma^2 x_2^2 Sin(x_1)Cos(x_1) + 2\alpha\gamma x_2 x_4 Sin(x_1)Cos^2(x_1) + \frac{\varepsilon}{\alpha}Sin(x_1)(\alpha\beta + \alpha^2 Sin^2(x_1))}{\alpha\beta + \alpha^2 Sin^2(x_1) - \gamma^2 Cos^2(x_1)} \\
&\quad - \frac{\gamma Cos(x_1)g}{\alpha\beta + \alpha^2 Sin^2(x_1) - \gamma^2 Cos^2(x_1)}u \\
\dot{x}_3 &= x_4 \\
\dot{x}_4 &= \frac{-\alpha\gamma x_4^2 Sin(x_1)Cos^2(x_2) - \gamma\varepsilon Sin(x_1)Cos(x_1) + \alpha\gamma x_2^2 Sin(x_1) - 2\alpha^2 x_2 x_4 Sin(x_1)Cos(x_1) + \alpha g u}{\alpha\beta + \alpha^2 Sin^2(x_1) - \gamma^2 Cos^2(x_1)}
\end{aligned}
\quad . \quad (1)
$$

Figure 10. A Furuta pendulum.

where $\alpha, \beta, \gamma, \varepsilon$, and g are constants, and $x_1 \sim x_4$ are the state variables, representing the angle of the horizontal arm, the angle of the vertical arm, and their derivatives. The DifferentialSystem actor in Ptolemy II allow us the enter the ODE in the form of (1), instead of wiring up individual integrators as in Figure 3[1]. The angleConversion composite actor[2] modulates the angle of the horizontal arm to $(-\pi, \pi]$. After verifying the correctness of the model with various inputs, such as step functions, we close the control loop with a linearization and state feedback for the stabilization case, as shown in Figure 11. An execution result is shown in Figure 12.

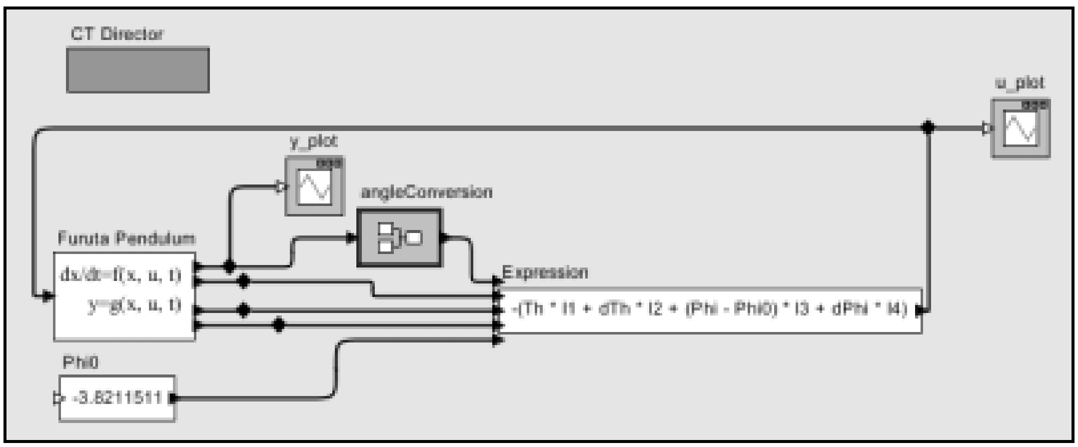


Figure 11. A continuous-time pendulum control model for stabilization.

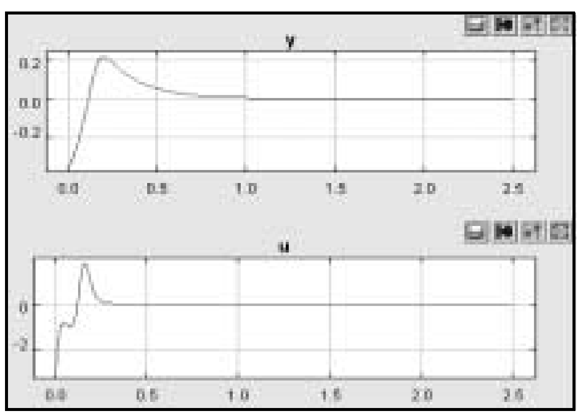


Figure 12. The simulation results for a continuous-time stabilization controller. Plot y shows the vertical angle of the upper beam, and plot u shows the control output.

1. This feature is called *higher-order actors*, which create other actors from declarative representation.
2. This is a *transparent* composite actor, which does not have a director but leverage the director on level up (the CT Director, in this case).

The next step is to discretize the continuous control law to get a sample-data discrete control law. Like in many paper designs, the discrete controller is considered to have no delay. Since the system is highly nonlinear, the closed-loop simulation capability comes handy. In terms of the design process, this step is primarily a combination of the following operations:

- refining the continuous controller interface into sampling, discrete controller, and zero-order hold specifications (this step is currently done in designer's mind or informally on paper);
- implementing the three specifications by executable components (in this case, choosing components from Ptolemy II actor libraries);
- composing the components with the continuous pendulum model, through CT and SDF models of computation;
- perturbing the sampling rate and the controller parameters to achieve desirable closed-loop control performance.

A Ptolemy II model at the end of this step is shown in Figure 13. The PeriodicSampler actor and the ZeroOrderHold actor convert between continuous signals and sampled data. The SDF domain inside the discrete controller composite actor executes entirely on discrete sequences of data. Note that many

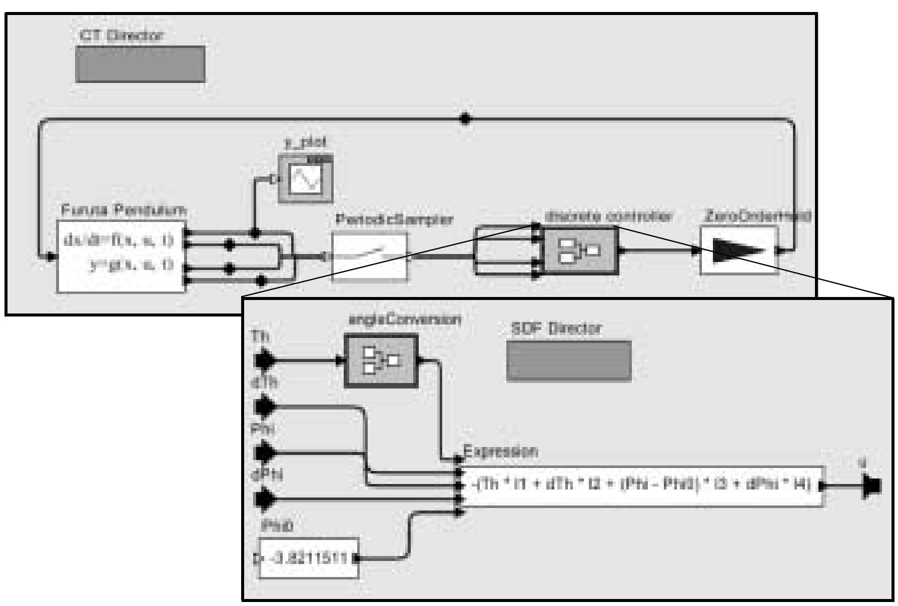


Figure 13. A hierarchical model for a discrete-time stabilization control.

components, including the pendulum model and the structure of the control law, are reused from the previous pure CT model. In particular, the angleConversion actor and the expression actor work both on continuous-time signals (in Figure 11) and discrete data (in Figure 13), a property we call *domain polymorphism*.

The stabilization controller only works when the pendulum already has a vertical/up initial position. To swing up the pendulum from a natural downward initial position, we use a modal controller with three discrete states, as suggested in [3]: *swingup*, *catch*, and *stabilize*.

To achieve this, a swingup controller and a catch controller are separately specified, implemented, and composed with the same pendulum model with different initial states. Like the stabilization controller, the catch controller is a state feedback controllers that tries to reduce the angle velocity of the vertical arm after it is swung up. The swingup controller, however, is an energy-based controller, which is started as a discrete design. In every sample step, the controller computes the energy of the pendulum and produces a control output that inject more energy into the pendulum system, so that it swings up. The three controllers are then composed through the FSM domain in Ptolemy II, as shown in Figure 14. This controller now replaces the discrete controller actor in Figure 13. Note that since the angleConversion is used in both modes, it is pulled out to a higher level than the state machine. Again, many of the components in previous designs are reused.

The model so far, although fairly complicated already, still has many idealized assumptions. For example, there is no computational delay, and sensing and actuating are instantaneous. The next steps in the design process will gradually add realistic concerns, like timing and precision.

We first consider execution delays of the control algorithm, which will be realized on an embedded processor with a real-time operating system (RTOS). The TM domain in Ptolemy II allows us to model the execution time of the controller, and its prioritized execution together with other actors on the same platform. Under the TM domain, we compose the idealized controller in Figure 14 with a Discard actor that models other tasks in the system, as shown in Figure 15. The Discard actor, triggered by a Poisson clock, discards the input data value, but affects execution time of the controller. Figure 16 (a) and (b) shows the differences before and after adding execution delays. The original control law, after adding delays, leads to an unstable system, and fails to swing up the pendulum. The delay can be compensated by deriving new

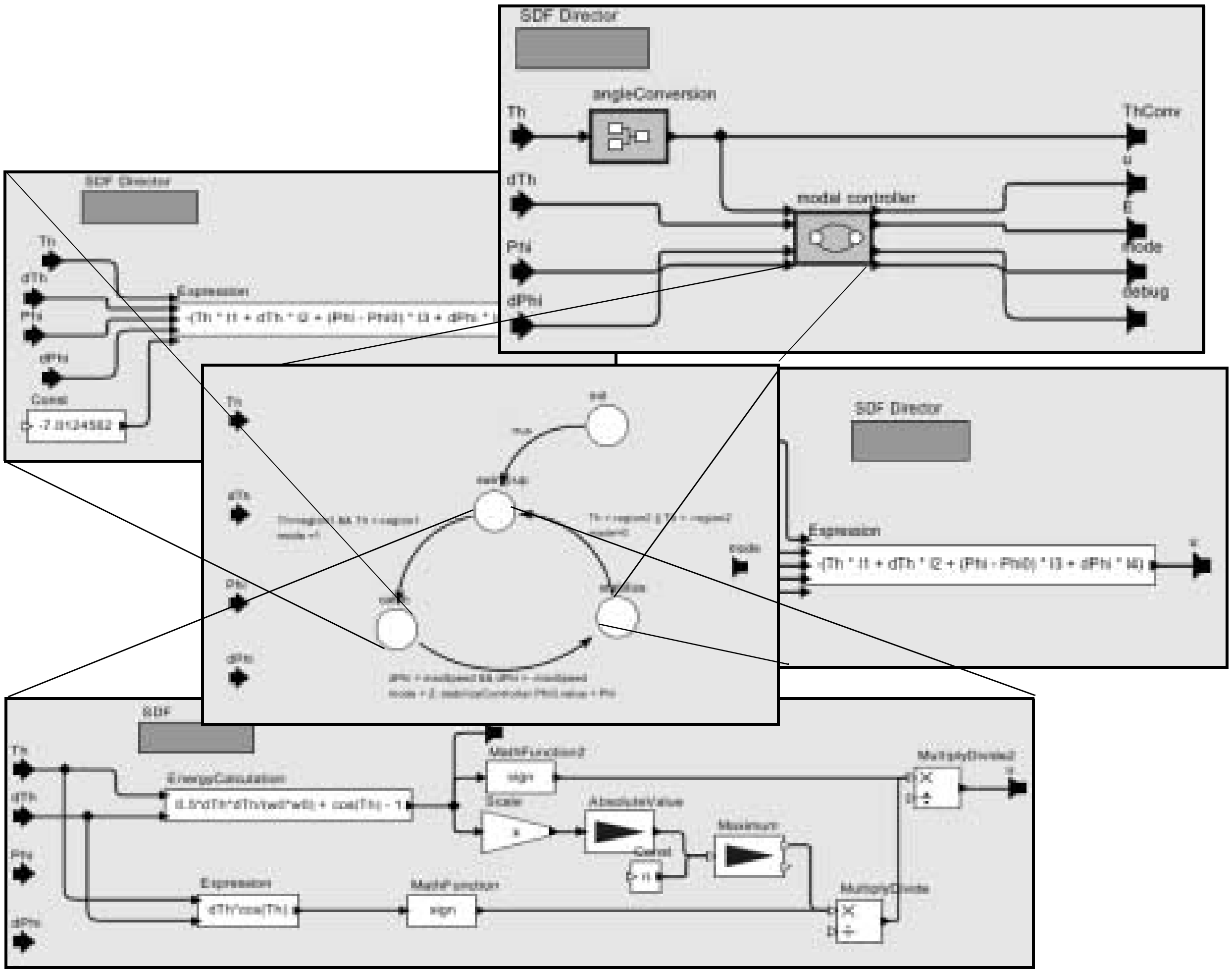


Figure 14. A multi-mode discrete-time controller for the swinging pendulum.

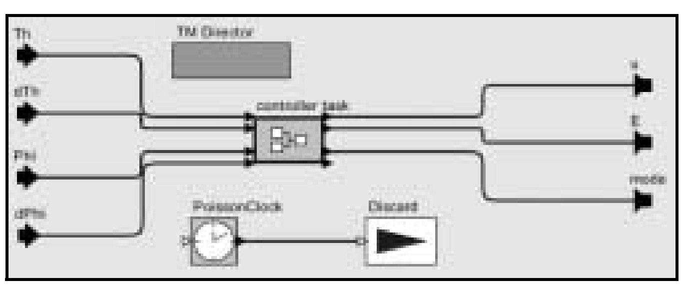


Figure 15. Modeling execution delays of the controller under prioritized multitasking execution.

(a).

(b).

(c).

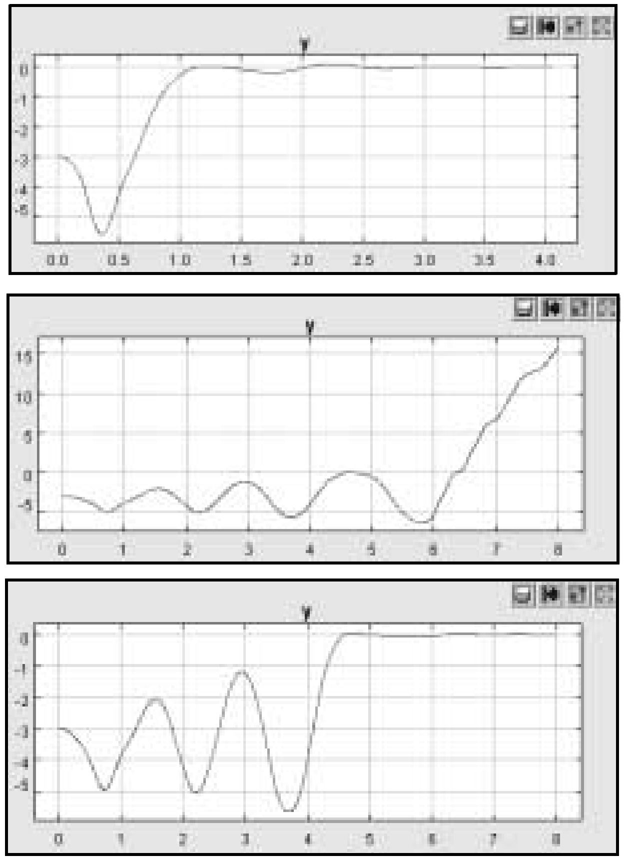


Figure 16. Execution results with and without delay: (a) original parameters without execution delay, (b) original parameters with execution delay, (c) compensated parameters with execution delay.

controller parameters. For example, using another set of parameter, the controller is again able to swing up the pendulum (Figure 16 (c)), but it now needs almost five seconds and several attempts to swing up the pendulum, and the performance decreases due to the delays.

Next, we consider a distributed implementation of the system, where the controllers is connected to the sensors and actuators through a network. At the network level, all communication packets can be treated as events, which may happen at any time. So, we use the discrete-event domain to model the message passing among actors that models the sensor, the actuator, and the controller. The sensor and the actuator are modeled as a pure time delay, indicating that they introduce a little delay during sensing and actuating. The network composite actor is internally implemented by pairs of transmitters and receivers and a shared communication media. Other devices on the network is modeled as a single node. Figure 17

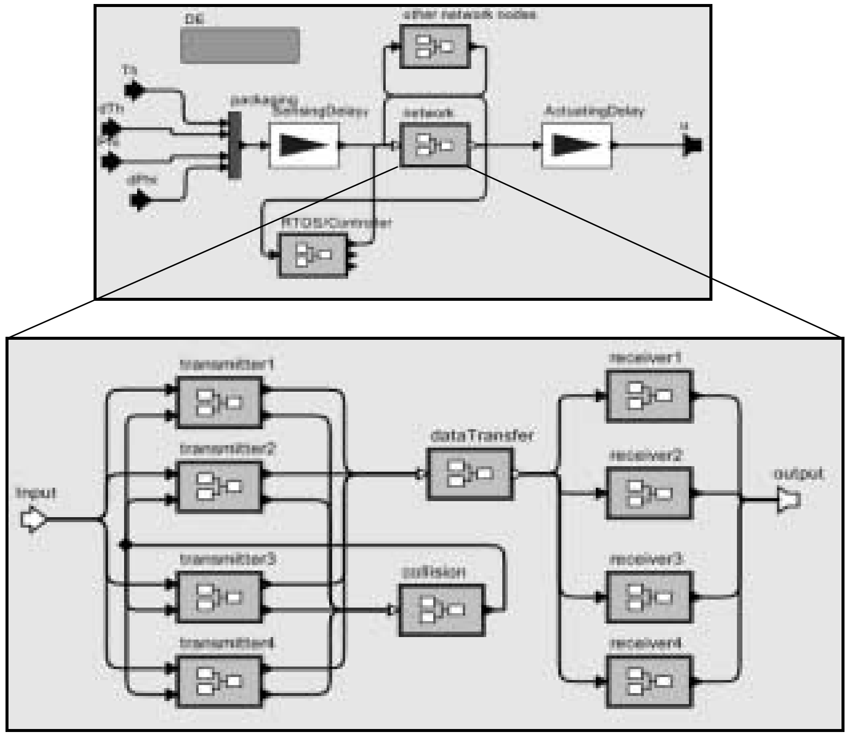


Figure 17. The network part of a distributed controller model.

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 outputs only the ones that are directed to the corresponding actor. Similarly to the discussion above, adding the communication network introduces more delays to the control path. This again requires an exploration of control laws and mode transition conditions.

Current efforts are in implementing hardware-in-the-loop simulation that enables realization transitions in the design process. A real controller and a real network, realize the embedded hardware and software, will replace the controller block at the top level of Figure 13. The dynamics of the pendulum together with sampling and zero-order hold, are still in the Ptolemy II model. The actor-oriented modeling framework defines a clear boundary for such integration.

6. Conclusion

This paper suggests an actor-oriented design methodology for developing complex control systems, and demonstrates its application using the Ptolemy II modeling and design environment. Actor orientation localizes interactions within the components of a system. Hierarchies of models of computation frameworks effectively manage heterogeneity in the system. Concepts of precise reactions and responsible frameworks are introduced to develop composable frameworks. An actor-oriented design can be carried out by step-by-step transitions of actors –– on their specification, implementation, and realization. A pendulum swinging-up control system is used as an example to illustrate the design methodology. Design concerns are gradually added to enrich the model and bring it close to realization. Closed-loop control performance can be simulated and quickly feedback to designers at each step.

Acknowledgement

This work is part of the Ptolemy project, which is supported by DARPA/ITO, the State of California MICRO program, and the following companies: Agilent, Cadence Design Systems, Hitachi, and Philips.

References

[1] G.A. Agha, “Concurrent Object-Oriented Programming,” *Communications of the ACM*, 33(9), pp. 125-141, 1990.

[2] G.A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, 1986.

[3] K.J. Aström and K. Furuta, “Swinging up a Pendulum by Energy Control,” *Automatica*, 36(2), pp. 287-295, February, 2000.

[4] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A, Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, B. Tabbara, *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*, Kluwer Academic Press, June 1997.

[5] G. Berry and G. Gonthier, “The Esterel synchronous programming language: Design, semantics, implementation,” *Science of Computer Programming*, 19(2):87-152, 1992.

[6] A. Benveniste and P. Le Guernic, “Hybrid Dynamical Systems Theory and the SIGNAL Language,” *IEEE Trans. on Automatic Control*, 35(5), pp. 525-546, May 1990.

[7] W. Boggs and M. Boggs, *Mastering UML with Rational Rose 2002*, Sybex, 2002.

[8] J.Davis II, C. Hylands, B. Kienhuis, E.A. Lee, J. Liu, X. Liu, L.Muliadi, S. Neuendorffer, J. Tsay, B. Vogel, and Y. Xiong, *Heterogeneous Concurrent Modeling and Design in Java*, Technical Memorandum UCB/ERL M01/12, EECS, University of California, Berkeley, March 15, 2001. (http://ptolemy.eecs.berkeley.edu/publications/papers/01/HMAD/)

[9] L. de Alfaro and T.A. Henzinger, “Interface Theories for Component-Based Design,” *Proc. of the First Workshop on Embedded Software*(EMSOFT 2001), Tahoe City, CA, LNCS 2211, Springer-Verlag, Oct. 2001.

[10] J. Eker, J.W. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, “Taming Heterogeneity—the Ptolemy Approach,” submitted to *Proceedings of the IEEE*.

[11] R. Esser, J.W. Janneck, “Moses - a tool suite for visual modeling of discrete-event systems,” *Proceedings of Symposium on Visual/Multimedia Approaches to Programming and Software Engineering* Stresa, Italy, September, 2001

[12] K. Furuta, M. Yamakita, and S. Kobayashi, “Swingup control of inverted pendulum using pseudo-state feedback,” *J. Systems and Control Eng.* 206, 1992, pp. 263-269.

[13] A. Girault, B. Lee, and E.A. Lee, “Hierarchical Finite State Machines with Multiple Concurrency Models,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 18(6), 1999.

[14] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[15]I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley Pub Co, 1999.

[16]I. Kumaran and S.I. Kumaran, *Jini Technology: An Overview*, Prentice Hall, 2001.

[17]A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi, “The Generic Modeling Environment,” in *Proceedings of IEEE International Workshop on Intelligent Signal Processing* (WISP’2001), May, 2001, Budapest, Hungary.

[18]E.A. Lee and T.M. Parks, “Dataflow Process Networks,” *Proceedings of the IEEE*, 83(5), pp. 773-801, May, 1995.

[19]E.A. Lee and D.G. Messerschmitt, “Synchronous Data Flow,” *Proc. of the IEEE*, 75(9), pp. 1235-1245, September, 1987.

[20]J. Liu, *Responsible Frameworks for Heterogeneous Modeling and Design of Embedded Systems*, Ph.D. Dissertation, EECS, University of California, Berkeley, Fall 2001.

[21]J. Liu and E.A. Lee, Timed Multitasking for Real-Time Embedded Software, submitted to IEEE Control Systems Magazine.

[22]J. Liu and E.A. Lee, “Component-based Hierarchical Modeling of Systems with Continuous and Discrete Dynamics,” *2000 IEEE Symposium on Computer-Aided Control System Design* (CACSD’00), Anchorage, Alaska, USA, September, 2000, pp. 95-100.

[23]N. Lynch, R. Segala, F. Vaandrager, and H.B. Weinberg, “Hybrid I/O automata,” in R. Alur, T.A. Henzinger, and E.D. Sontag, editors, Hybrid Systems III, volume 1066 of Lecture Notes in Computer Science. Springer-Verlag, 1996, pp. 496-510.

[24]N. Halbwachs, Synchronous Programming of Reactive Systems, Kluwer Academic Publishers, Boston, 1992.

[25]J. Paunicka, B. Mendel, and D. Corman, "The OCP - an Open Middleware Solution for Embedded Systems," in Proceedings of the American Control Conference, Arlington, VA, 2001, pp. 3345-3350.

[26]B. Selic, G. Gullekson, and P. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York, NY, 1994.

[27]R. Sessions, *COM and DCOM: Microsoft's Vision for Distributed Objects*, New York, NY: John Wiley & Sons, 1997.

[28]D.B. Stewart, R.A. Volpe, and P.K. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," *IEEE Trans. on Software Engineering*, 23(12), 1997, pp.759-776.

[29]C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley Pub., Jan. 1998.

[30]*Common Object Request Broker Architecture*, Object Management Group (OMG), July, 1995.

[31]*Dome Guide*, Honeywell Inc. 1999. (http://www.htc.honeywell.com/dome/).