Actor-Oriented Control System Design: A Responsible Framework Perspective

Jie Liu, Johan Eker, Jörn W. Janneck, Xiaojun Liu, and Edward A. Lee *Fellow, IEEE*

Abstract

Complex control systems are heterogeneous, in the sense of discrete computer-based controllers interacting with continuous physical plants, regular data sampling interleaving with irregular communication and user interaction, and multilayer and multimode control laws. This heterogeneity imposes great challenges for control system design 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 tackles 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). Semantically different frameworks are composed hierarchically to manage heterogeneous models and achieve actor and framework reuse. We introduce a notion of responsible frameworks to characterize the property that a framework can aggregate individual actor's execution into a well-defined composite execution such that heterogeneous models can be composed.

This methodology is implemented in the Ptolemy II software environment. We discuss how some of the most useful models for control system design are implemented as responsible frameworks. As an example, the methodology and the Ptolemy II software environment is applied to the design of a distributed, real-time software implementation of a pendulum inversion and stabilization system.

Index Terms

Control system design methodology, heterogeneous modeling, hierarchical heterogeneity, responsible frameworks, actor-oriented design, Ptolemy II

I. INTRODUCTION

Embedded control system design is a complex and error prone task, not only because the algorithms implemented in these systems contain domain-specific expertise, but also because these systems are

Manuscript received March 31, 2002; revised March 21, 2003. This work is supported in part by the Ptolemy project, which is supported by DARPA, the National Science Foundation, the State of California MICRO program, and the following companies: Agilent, Atmel, Cadence Design Systems, Hitachi, National Instruments, and Philips.

Jie Liu is with Palo Alto Research Center (PARC), 3333 Coyote Hill Road, Palo Alto, CA 94304. Email: jieliu@parc.com

Johan Eker is with Ericsson Mobile Platforms AB, 221 83 Lund, Sweden. Email: Johan.Eker@emp.ericsson.se

Jörn W. Janneck, Xiaojun Liu, and Edward A. Lee are with EECS Department, University of California, Berkeley, CA 94720. Email: {janneck, liuxj, eal}@eecs.berkeley.edu

heterogeneous. Control systems interact with the physical world. The nature of the problems requires that the embedded computer systems be reactive, real-time, non-terminating, and collaborative. Computer-based control systems 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 ways, such as closed-loop control performance analysis, design modularity and understandability, component reuse, testing, and debugging.

A classical control system design process is divided into stages. Working from a conceptualized description of the control goal, control engineers derive control algorithms based on high-level plant models, an idealized execution platform, and simplified performance requirements. The control laws are usually validated by a combination of mathematical proofs and continuous-time/mixed-signal simulations. In the next stage, implementation teams take the control algorithms, typically in forms of formulas on paper and simulation results, and start to design system architectures and embedded software. This team 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 hardware 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 performances and their robustness.

The development relies on the integrated testing phase following this design cycle. Errors found at this point usually indicate that some implementation decisions violated the design assumptions, but it is often very hard to localize either of them. This makes the design process time-consuming and fragile: a slight change of the control specification may require a complete cycle of redesign.

One problem is the distinct expertise required in various stages and the different modeling paradigms used in each stage and each subsystem. Not many control engineers have sufficient software engineering experience, and not many software engineers have sufficient experience with control system design. 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 illustrated in Figure 1, where the control design team and the system design team both work on the same model. Design decisions are gradually introduced into the model through iterations, such that it is refined step-by-step toward final implementation. Changes in the implementation structure are reflected in the control performance and, similarly, modifications to the

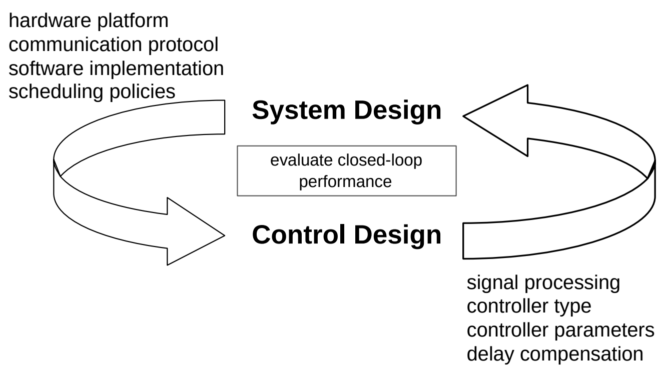

Fig. 1. In a more integrated design flow, control designs and system implementation are tightly coupled, so that design decisions can be quickly evaluated and fed back between phases.

control laws are reflected in the software implementation. 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 (such as hybrid systems [1]) that integrate more than one kind of dynamics, but also need design methodologies and tools that help designers decompose a system into modularized and domain-specific subsystems and (re-)compose these components into a coherent and realistic system-level model.

This paper presents an *actor-oriented* design methodology for complex control systems. Actor orientation separates the functionality concerns (modeled as actors) from the component interaction concerns (modeled as frameworks), and gives well-defined scopes for model refinement and system realization. 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. We introduce a notion of *responsible frameworks* as a characterization of hierarchically composable frameworks. Responsible frameworks guarantee that the execution of an actor, once started, can always reach a quiescent state, and thus provide a coarse-grained atomicity of execution and well-defined system states. Furthermore, a responsible framework itself, together with the actors under its control, can be treated as a single component at a higher level of hierarchy, thus the atomicity of execution can be composed. Using Ptolemy II [2] as a concrete design environment, we show how some of the widely used models of computation for control system design — continuous time(CT), discrete event(DE), synchronous dataflow(SDF), timed multitasking(TM), and finite state machine(FSM) — can be implemented as responsible frameworks.

The remainder of the paper is organized as follows. Section II describes the actor-oriented design methodology with a motivation of hierarchical heterogeneity. Section III introduces the notion of responsible frameworks using a labeled transition system model of actors and frameworks. In section IV, we present the Ptolemy II design environment that supports an actor-oriented design methodology, and discuss the implementation of some models of computation as responsible frameworks. An example of designing a software control system for inversion and capture of an inverted pendulum is given in section V. The example is carefully chosen to include key features of real-world problems, but to be simple enough to permit a complete description.

II. 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 [3]. An 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 [4] to formally specify systems. Object-oriented software environments, such as Rational Rose [5], GME [6], and DOME [7], have been applied in control system designs.

Noticing that some objects usually work together to provide a coherent piece of functionality, middleware-oriented design advocates the encapsulation of one or more objects into conceptual *services*, and the composition of 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 designs. Thus, they appear more often in large-scale applications, which leverage distributed object infrastructures, such as CORBA [8], DCOM [9], and JINI [10]. The open control platform (OCP) [11] developed at Boeing is an example of middleware-oriented design for real-time control systems.

Despite their conceptual differences, the basic structure in 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 resource utilization, 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 correctness of component composition is left to designers.

Actor-oriented designs acknowledge the variety of interaction models among components, and express these interaction styles independently from the functionality of components.

A. Actor Orientation

An *actor* is an encapsulation of parameterized actions performed on input data to produce output data. Actors 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 OO designs, does not have to have a call-return semantics. Essentially, an actor defines local activities without referencing of other actors.

There are many examples of actor-oriented design environments, including Simulink from MathWorks, LabVIEW from National Instruments, SPW from Cadence, Cocentric studio from Synopsys, and ROOM (Real-time Object-Oriented Modeling [12]) from Rational Software (now IBM). In the academic community, active objects and actors [13], [14], port-based objects [15], hybrid I/O automata [16], Moses [17], Polis [18], Ptolemy [19] and Ptolemy II [2] all emphasize actor orientation.

The behaviors of a set of actors are not well-defined without a coordination model. A *framework* is an environment that actors reside in, and it defines the interaction among actors. Frameworks also differentiate many actor-oriented modeling paradigms. For example, ROOM and Agha's actor models suggest that actors be "active;" that is, each of them has its own thread of control. Some others, such as 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. A framework implements a model of computation. 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, and SPW is a dataflow framework. But, for complex control systems, a single MoC is usually not enough.

Figure 2 shows an actor-oriented 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 maps well to the actual partition of a system, and it helps to identify concurrency and communication issues. For example, the plant operates concurrently with the controller in the physical world, and within the controller, the three actors implemented by embedded software may operate sequentially because of data dependencies. However, the interaction styles among components are not obvious in this view. There is no immediate distinction that the plant, sensors, actuators, and the controller interact in a continuous-time/mixed-signal style, while within the controller, a dataflow model may be more appropriate.

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, whereas the dynamics of the controller is discrete. In more complicated cases, even within the controller, the dynamics of control

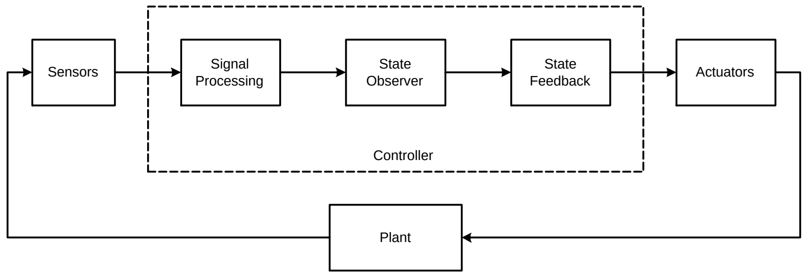

Fig. 2. An actor-oriented view of a control system.

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 these separate areas are relatively well understood and established, the integration of these dynamics brings significant complexity to the design problem. If a design environment only supports a single MoC, then it can only model certain part of a complex system, or at a certain level of abstraction. The correctness of a final design has to rely on final integrated testing, which typically leads to long design cycle and high cost.

B. Hierarchical Heterogeneity

A powerful concept to scale up actor-oriented designs is *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 that hides the detail of a subsystem from the rest of the system. It also defines interaction scopes so that modification in a 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 of MOC, an approach called *hierarchical heterogeneity* [20]. 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 may allow certain parts of the system to be correct by construction, because of the formal properties obtained by that specific MoC.

Actor encapsulation and hierarchical heterogeneity are powerful techniques that allow designers to work on different levels of abstraction but still keep a global view of the system. For example, Figure 3 shows a hierarchical model for the control system illustrated in Figure 2. At the beginning of a design, control engineers can work on a model completely in the continuous-time domain, like the top part of Figure 3.

After obtaining the continuous control law, the software team can *refine* and *replace* the controller actor by a dataflow model that is more suitable for software implementation. A design environment that supports hierarchical heterogeneity will allow the software team to continue to use the same plant model to verify the software decisions.

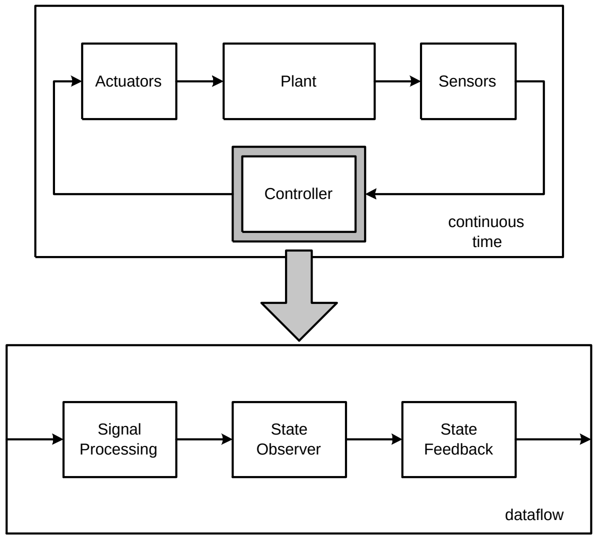

Fig. 3. A hierarchical view of the control system in Figure 2

Building hierarchically composable heterogeneous MoC frameworks is not trivial. First of all, as the coordinator among actors, a framework needs to constrain and control the execution of individual actors so that their overall behavior obey the specified model of computation. More importantly, a framework needs to aggregate actor executions into a coarse-grained atomic execution so that at a higher level of hierarchy, the entire framework can be viewed as an opaque actor. In the next two sections, we describe the Ptolemy approach of implementing hierarchical heterogeneity, building on a notion of responsible frameworks.

III. Responsible Frameworks

The interaction among actors and frameworks can be formulated in a labeled transition system model.

A. Actors and Frameworks

A *framework* Φ is a labeled transition system $\Phi = (\Sigma, \Lambda, \Gamma, \Omega)$, where

– Σ is a set of framework states;

– Λ is a set of actions;

– $\Gamma \subseteq \Sigma \times \Lambda \times \Sigma$ is a set of transitions guarded by the actions in Λ; for a transition $(\sigma_1, \lambda, \sigma_2) \in \Gamma$, we also write $\sigma_1 \xrightarrow{\lambda} \sigma_2$.

– Ω is a set of initial states.

We sometimes write $\Phi = (\Sigma_\Phi, \Lambda_\Phi, \Gamma_\Phi, \Omega_\Phi)$ if distinguishing the frameworks is important. The state of a framework reflects the information shared among actors in the framework, such as the notion of iterations or time and the communication semantics. For example, if the communication channels among actors are FIFQ queues, then these queues are part of the framework state.

In the set of actions, we explicitly define NOP, $\epsilon \in \Lambda$, which enables no transitions or only stutter transitions; i.e. $(\sigma_1, \epsilon, \sigma_2) \in \Gamma \Rightarrow \sigma_1 = \sigma_2$.

An *actor* A in a framework Φ is a tuple $A = (S, T, I, Q)$, where

– S is a set of actor states;

– $T \subseteq S \times \wp(\Sigma_\Phi) \times \Lambda_\Phi \times S$ is a set of transitions, where $\wp(\Sigma_\Phi)$ is the power set of Σ_Φ.

– $I \subseteq S$ is a set of initial states;

– $Q \subseteq S$ is a set of quiescent states. Typically $I \subset Q$.

In a transition $(s, G, \lambda, s') \in T$, also written as $s \xrightarrow{G/\lambda} s'$, $G \subseteq \Sigma_\Phi$ is called the *guard* of the transition, and $\lambda \in \Lambda_\Phi$ is call the *action* of the transition.

The execution of a composition of a framework and a set of actors has a synchronous, interleaving semantics. At any step, an actor makes a transition if the current state of the framework is in the guard of the transition. The action of the actor transition may change the state of the framework. That is, a transition $s \xrightarrow{G/\lambda} s'$ is taken only if the current state of the framework $\sigma \in G$. Futhermore, for any $\lambda \in \Lambda$ with $s \xrightarrow{G/\lambda} s'$, there exist $\sigma \in G$ and some $\sigma' \in \Sigma$, such that $\sigma \xrightarrow{\lambda} \sigma'$. In general, neither the framework nor the actors have to be deterministic. In cases where multiple actor transitions are enabled by the framework state, they are executed nondeterministically.

For any $t = (s, G, \lambda, s') \in T$, we define functions $Src(t) = s$ and $Des(t) = s'$, which give the source state and the destination state of the transition. An *execution path* $P_{s,s'}$ of an actor from state s to state s' is a chain of transitions, i.e. a finite sequence $\{t_1, t_2, ..., t_n\} \subseteq T$ satisfying $Src(t_1) = s$, $Des(t_n) = s'$, and $\forall 1 \leq i \leq n-1, Des(t_i) = Src(t_{i+1})$. Because of the nondeterminism of the framework and the actors, there may be more than one path from s to s'. We say that $P_{s,s'}$ goes through state s'' if there are two transitions in the path such that s'' is the destination of one and the source of the other.

A key to achieving hierarchical heterogeneity is to hide intermediate actor execution and expose only "significant states" to the outside. For this purpose, we define a set of *quiescent states* which is a subset of the state space of actors. In theory, the set of quiescent states can be any subset. In practice, these notions are typically well defined, such as the state at the termination of a functional block, the state at the completion of a transaction, the state of the subsystem at a particular time, or states that can be represented by a minimum number of variables.

The execution of an actor alternates between two phases: a quiescent phase and an execution phase. The execution enters the quiescent phase when the actor reaches a quiescent state. The first transition out of a quiescent state is of special interest since it indicates that the actor enters a new execution phase. For this reason, we give these transitions a special name: for a quiescent state $q \in Q$, a transition t with $Src(t) = q$ is called a *trigger* at q, denoted by t_q. All triggers at q form a set T_q.

B. Responsible Frameworks

Ideally, one would like that an execution phase can eventually reach another quiescent state, so that from a high level view, the execution can be abstracted into an atomic transition from one quiescent state to another. Formally, a *precise reaction* $P_{q,q'}$ is an execution path from $q \in Q$ to $q' \in Q$ that does not go through any quiescent states. So, if the state trajectory of the framework enables all transitions in $P_{q,q'}$, then a precise reaction can be achieved. However, since the evolution of framework states depends not only on the framework, but may also on the execution of all the actors in the framework, the reasoning about a precise reaction can be nontrivial.

In general, collaborations between actors and framework are essential to obtain precise reactions. A trigger t_q is *responsible* if it can guarantee that for all possible future states of the framework, subject to the model of computation the framework implements, the execution starting from q will reach some $q' \in Q$. A framework is *responsible* if, given the responsible triggers from the actors, it can guarantee precise reactions of all actors.

One particular interaction pattern between actors and a framework is shown Figure 4, where an explicit trigger action, labeled as A:trigger is introduced for an actor A to signal the framework Φ for possible precise reactions. The framework may choose to accept the trigger by making a transition that allows the actor to proceed its execution. The choice is modeled as nondeterministic framework transitions such that the framework may also go to a state to prevent the actor from execution. This pattern is widely used in practice and all MoCs in section IV implement it.

To fully achieve responsibleness, after a framework accepts a responsible trigger, it needs to control the framework state transitions to maintain the conditions that the actor assumed for the responsible trigger. This property is very hard to analyze in general, however, it is much simpler if the framework implements a formal model of computation and constrains the interaction patterns among actors. To explain this further, we leverage the separation of computation and communication in the actor models. A transition $s \xrightarrow{G/\lambda} s'$ of an actor is an *internal transition* if $G = \Sigma$ and $\lambda = \epsilon$. That is, the transition can be taken at any time, and it has no effect on the framework state. Otherwise, the transition is called an *I/O transition*.

A communication model defines the guards and actions on I/O transitions. For example, suppose that the communication from actor A to B in Figure 5 is a FIFO queue with infinite capacity. Then output

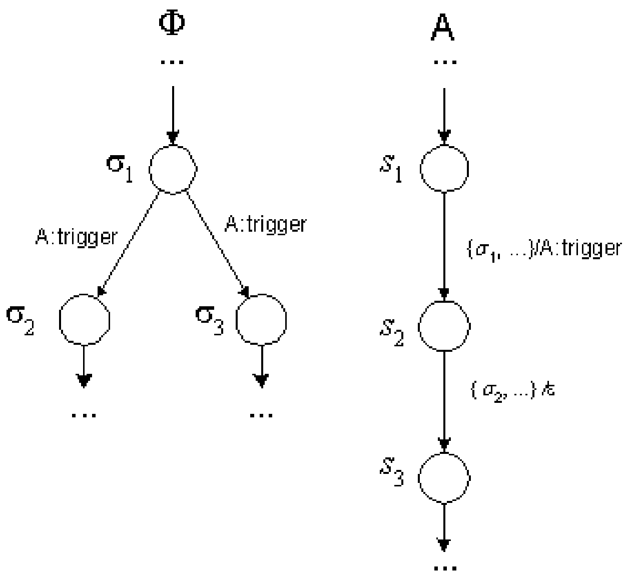

Fig. 4. An interaction pattern to achieve responsible frameworks. The framework consults the actor for potential responsible triggers.

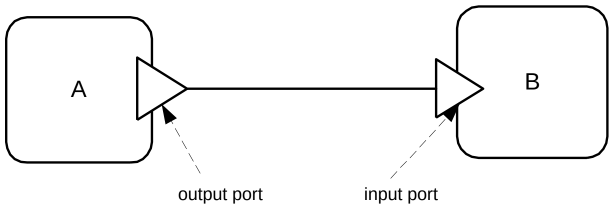

Fig. 5. Two actors communicate through ports and a framework.

transitions of actor A are always enabled. Thus, an output transition t has Σ as the guard and has an action that changes the framework state by adding one more data token to the end of the communication queue. The guard of the corresponding input transition, however, contains the framework states where there is at least one token in the queue. Notice that adding more tokens into the queue will not disable the transition, so the executions of the two actors connected by the queue are essentially decoupled. Consider further that $P_{q,q'}$ is a precise reaction of B, and there are n input transitions in the path. If a trigger t_q guarantees that there are at least n tokens in the queue, then the execution following that trigger can always reach state q'. Such a t_q is a responsible trigger. If in a framework, all communications are FIFO queues, and the triggers are responsible, then the framework is responsible by simply implementing the triggering pattern in Figure 4. This framework implements a dynamic dataflow (DDF) MoC.

Not all frameworks are responsible. In the above example, if the trigger at q in actor B, instead of testing for the existence of n input tokens, has a guard $G = \Sigma$, then it is not a responsible trigger. This framework, which may implement a Kahn-MacQueen process network MoC [21], is not a responsible framework. For another example, suppose that in Figure 5, the communication between A and B has a *rendevouz* semantics, where the input transition in B has a guard requiring that there be an output port ready to produce data and the output transition in A has a guard requiring that there be an input port ready to receive data. However, there is no way for a trigger at an early state q to predict whether there will be another actor ready for rendevouz. Thus, at least one of the triggers in A or B has to be irresponsible to make any execution of the model. This framework, which may implement communicating sequential process (CSP) [22], cannot be a responsible framework.

Notice that responsible frameworks do not prevent deadlocks. In fact, it is possible that at some framework state, none of the responsible triggers are enabled, such that none of the actors can execute. However, they do guarantee that even in a deadlock state, all actors are in their quiescent states, thus the system as a whole has a well-defined quiescent state.

C. Compositional Precise Reaction

A composition of a framework and the actors under its control is called a *composite actor*. In order for a composite actor to communicate with other actors at a higher level, more states and transitions need to be added to the framework. We call such a framework an *open framework*, denoted by $\hat{\Phi} = (\Sigma_{\hat{\Phi}}, \Lambda_{\hat{\Phi}}, \Gamma_{\hat{\Phi}}, \Omega_{\hat{\Phi}})$.

An open framework should be compatible with the original framework. Formally, let Σ^+_{Φ} be the additional states for an open framework $\hat{\Phi}$ spanned by a set of I/O variables, Ω^+_{Φ} be the initial states for these variables, then $\hat{\Phi}$ satisfies:

- $\Sigma_{\hat{\Phi}} = \Sigma_{\Phi} \times \Sigma^+_{\Phi}$
- $\Lambda_{\Phi} \subset \Lambda_{\hat{\Phi}}$
- $\Gamma_{\hat{\Phi}}$ is compatible with Γ_{Φ}. That is, if $\sigma_{\Phi} \xrightarrow{\lambda} \sigma'_{\Phi}$ in Φ, then for any $\sigma^+_{\Phi} \in \Sigma^+_{\Phi}$, $(\sigma_{\Phi}, \sigma^+_{\Phi}) \xrightarrow{\lambda} (\sigma'_{\Phi}, \sigma^+_{\Phi})$ in $\hat{\Phi}$.
- $\Omega_{\hat{\Phi}} = \Omega_{\Phi} \times \Omega^+_{\Phi}$.

For a composite execution, additional I/O transitions are needed to interact with the outside framework. Let C be a composite actor implemented by $\hat{\Phi}$ and a set of actors $A_i = (S_i, T_i, I_i, Q_i)$, $i \in \{1, ..., n\}$, and $\Phi' = (\Sigma_{\Phi'}, \Lambda_{\Phi'}, \Gamma_{\Phi'}, \Omega_{\Phi'})$ be the external framework for C. Then, C is a tuple (S_C, T_C, I_C, Q_C) satisfying the following constraints:

- S_C is a set of states. $S_C = \Sigma_{\hat{\Phi}} \times \prod_{i=1}^{n} S_i$.
- T_C is a set of transitions, such that:
  - All interactions between $\hat{\Phi}$ and A_i are internal transitions for C. That is, if $\exists s_i, s'_i \in S_i$, $G \subset \Sigma_{\hat{\Phi}}$, and $\sigma_{\hat{\Phi}} \in G$, such that $(\sigma_{\hat{\Phi}}, \lambda, \sigma'_{\hat{\Phi}}) \in \Gamma_{\hat{\Phi}}$ and $(s_i, G, \lambda, s'_i) \in T_i$, then for any $s_c = (\sigma_{\hat{\Phi}}, s_1, ..., s_{i-1}, s_i, s_{i+1}, ..., s_n)$ and $s'_c = (\sigma'_{\hat{\Phi}}, s_1, ..., s_{i-1}, s'_i, s_{i+1}, ..., s_n)$, $(s_c, \Sigma_{\Phi'}, \epsilon, s'_c) \in T_C$.

  - I/O transitions added to change the states in $\Sigma_{\hat{\Phi}}$ should not change the states in S_i or Σ_{Φ}. That is, if $(s_c, G_{\Phi'}, \lambda_{\Phi'}, s'_c) \in T_C$ with $\lambda_{\Phi'} \in \Lambda_{\Phi'}$, $G_{\Phi'} \neq \Sigma_{\Phi'}$, and $s_c = (\sigma_{\Phi}, \sigma^{+}_{\Phi}, s_1, ..., s_n)$[1] then $s'_c = (\sigma_{\Phi}, {\sigma^{+}_{\Phi}}', s_1, ..., s_n)$, for some ${\sigma^{+}_{\Phi}}' \in \Sigma^{+}_{\Phi}$.

- $I_C = \Omega_{\hat{\Phi}} \times \prod_{i=1}^{n} I_i$
- $Q_C \subseteq \Sigma_{\hat{\Phi}} \times \prod_{i=1}^{n} Q_i$

So, I/O transitions for a composite actor are completely separated from the interaction between the framework and the actors. In this way, the inside of a composite actor will still obey the model of computation the framework implements, and the interaction at the boundary can be studied between frameworks without involving the behaviors of individual actors.

A key property for responsible frameworks is that precise reactions of individual actors can be aggregated into a precise reaction of the composite actor. As defined above, at a quiescent state of a composite actor, all internal actors must be at their quiescent states. For a responsible framework, this is relatively easy to achieve, since all executions from quiescent states of individual actors will reach another quiescent state in a finite number of transitions.

Another important issue for building compositional precise reactions is to define responsible triggers for the composite actor. Generally, how many individual precise reactions to aggregate into one composite precise reaction could be domain dependent. In practice, time and data dependencies usually play a central role. We will discuss them more specifically in the next section.

IV. Practice in Ptolemy II

Ptolemy II is a graphical modeling and design environment implementing an actor-oriented design methodology and hierarchical heterogeneity.

The basic building blocks in a Ptolemy II model are *atomic actors*. Atomic actors encapsulate basic computation, from simple arithmetic operations to more complex ones like an FFT. Actors have input and output ports and can be composed by connecting corresponding ports. A composition of actors is guided by a *director*, which represents a model of computation. In Ptolemy II, a model of computation is also called a *domain*. A director may control the execution of actors through an `Executable` interface[2]. The communication mechanisms among actors are implemented by *receivers*, contained by input ports. A director, together with all receivers, defines a framework. To obey a specific model of computation, a director and receivers must match.

[1]Here we expand $\sigma_{\hat{\Phi}} = (\sigma_{\Phi}, \sigma^{+}_{\Phi})$, similarly for s'_c.

[2]For certain frameworks, such as PN or CSP discussed in section III-B, where the triggers are always irresponsible, an actor can also be active, having its own thread of control.

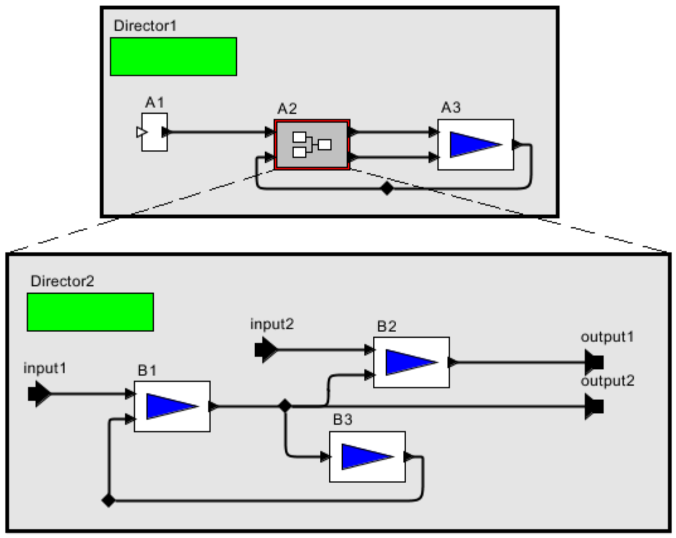

Fig. 6. A hierarchical model in Ptolemy II.

A model is a hierarchical composition of actors, for example, shown in Figure 6. Atomic actors, such as `A1` and `B1`, appear at the bottom of the hierarchy. Composite actors, such as the `A2`, can be further contained by other composite actors, so the hierarchy can be arbitrarily nested. Hierarchical heterogeneity is achieved by having different directors at different levels of the model.

Directors and receivers, when possible, are carefully implemented as responsible frameworks. A precise reaction of atomic actors is achieved by defining a multi-phase execution of actors. While an actor has a `fire()` method implementing its main functionality, it also provides a `prefire()` method that, with the help of receivers, may be used to test for responsible triggers. To implement the trigger pattern in Figure 4, a director first invokes the `prefire()` method, and then fires the actor only if the `prefire()` method returns true. Compositional precise reaction is achieved through a notion of *iteration* defined for each domain. A firing of a composite actor executes an iteration of the contained subsystem, and the `prefire()` method of the composite actor delegates to its local director to compute whether an iteration can be finished. In this section, we discuss some of the most useful models of computation for control system design and describe how responsible frameworks and compositional precise reactions are implemented for them.

A. Dataflow models

In dataflow models [23], connections represent data streams, and actors compute their output data streams from input streams. The execution order of the actors is only constrained by the data dependency among them. This makes dataflow models amenable to optimized execution, for example to minimize memory requirements, 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) [24] is a particularly restricted special case. A precise reaction of an actor in the SDF framework consumes a fixed number of data tokens from each input port, and produces a fixed number of tokens to each output port. The framework state is the number of tokens in the FIFO queues that mediate the communication among actors. For a composite SDF model, an iteration is required to start and end in the same framework state, so that it can be repeated indefinitely in bounded memory and without deadlock. Whether a precise reaction exists for a given SDF model can be statically determined, and if it exists, it can be composed as a fixed sequence of precise reactions of the component actors. The SDF scheduler in Ptolemy II performs this analysis. For algorithms with a fixed structure, SDF can produce very efficient schedules.

An iteration of an SDF model is a precise reaction of the model.

B. Continuous time

Continuous-time (CT) models, such as ordinary differential equations (ODEs) and differential algebraic equations (DAEs), are widely used in control system designs for modeling physical dynamics and continuous control laws. The state of a CT system is stored in continuous state variables represented by *integrators*. An ODE can be modeled as a feedback of integrators with stateless functional actors. Algebraic equations themselves are stateless.

A significant meaning that a CT framework imposes is the notion of *time*, and the state of a CT model evolves with respect to that. The execution of a CT model computes a numerical solution for the differential equations at a discrete set of time points. A CT framework, especially when interacting with discrete models, needs to carefully control the progression of time to comply with system causality and numerical accuracy.

After choosing a numerical integration step size, the entire CT model is reduced to an algebraic system at that time instant, and the responsible triggers for each actor can simply be achieved through analyzing data dependency. Noticing that there are no system states being stored in the communication channels from one time instant to the next, a CT receiver is implemented simply as a buffer that holds one data token and can be overwritten by new tokens.

An iteration in a CT model is defined as successfully resolving the system state and producing the outputs at a particular time instant. Although a numerical integration method can be fairly complicated, involving firing actors multiple times within an iteration, these intermediate steps are not seen from the outside of a CT composite actor. When contained by discrete, timed domains (such as the discrete event domain), where there could be two notions of time across the hierarchy, a compositional responsible trigger is achieved by examining the time difference and possibly performing a rollback if the causality constraints between models are violated. A detailed discussion on this topic can be found in [25].

C. *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.

To implement a DE framework, a global event queue is used to sort events into a chronological order. Receivers in this domain are proxies for actors to put events into the global event queue. The DE director manages the notion of time and takes the first event from the queue to trigger the corresponding actor. The execution of an actor is only triggered when there is an event for it. An actor may provide a responsible trigger by examining its internal requirements and the value of the event. For example, if the actor is internally a dataflow actor that requires more than one event to complete an iteration, then the composite actor can refuse to fire until enough tokens are accumulated.

An iteration of a DE domain is defined as the processing all events at a particular time stamp. This notion is consistent with the CT domain in the sense that time only progresses between iterations.

D. *Timed Multitasking*

The timed multitasking (TM) model [26] is a programming model for time deterministic multitasking embedded software. As a simulation model, it also 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 modeled.

The execution model for the TM domain is not very different from that of the DE domain. In fact, TM receivers are inherited from DE receivers to serve as proxies for the global event queue. However, in addition to sorting events in their chronological order, simultaneous events are also sorted by their priorities. Conceptually, an actor is triggered only when the trigger event has the highest priority among all current events.

In the TM domain, an actor can specify its execution time, which is the amount of time it needs to finish its execution once triggered. The output of the actor is produced only when the simulated time reaches the execution finish time. To simulate preemptive execution, an actor is *not* actually executed when it receives a trigger event. In fact, the progression of time is monitored, such that the actor is only fired when its execution finish time is reached.

An iteration of a TM domain also corresponds to processing all events at a time stamp. However, depending on the actors' execution time, not all iterations produce new events.

E. Modal Models

Modal models are built using the finite state machine (FSM) domain. FSMs are used extensively in designing sequential control logic and operation modes. FSM models are amenable to in-depth formal analysis and verification. In Ptolemy II, they can be used to specify the behavior of atomic actors, or they can be hierarchically composed with other models of computation as modal models.

When the behavior of an atomic actor is specified by a state machine, it can be straightforwardly mapped to the actor definition given in section III-A. All states are quiescent. A precise reaction consists of reading at most one token from each input port, evaluating the transitions from the current state, making an enabled transition if one exists, and producing output tokens as specified by the actions on the enabled transition.

Modal models extend Statecharts [27] by allowing FSMs to be hierarchically composed with a variety of models of computation. For example, hybrid systems are modal models that compose the FSM and CT domains. A modal model consists of a state machine that captures the modes and mode transitions, and mode refinements, which are composite actors specifying the behavior in each mode. Responsible frameworks make it possible for composite actors to serve as refinement for state machines. Compositional precise reactions allow the system to have a well-defined quiescent state when mode switching occurs between iterations.

A precise reaction of a modal model consists of evaluating the discrete transitions from the current mode, conditionally executing one iteration of the refinement of the current mode, and making an enabled mode transition. All refinements of a modal model must support well defined iterations that start and end in quiescent states of the refinements. The modal model framework maintains the invariant that all refinements are quiescent when the model is in the quiescent phase.

V. EXAMPLE

The inverted pendulum is a classic control problem that has been extensively studied in the literature. However, as a control problem, these studies typically stop at designing a continuous or discrete-time control law, without touching the implementation issues.

The inherent instability of the inverted pendulum makes it a suitable demonstrator for real-time control software, since only if the timing of the controller is correct and the latency is short will it be able to control the pendulum. In this section, we use a pendulum controller implemented in real-time software on a distributed platform to demonstrate how an actor-oriented design methodology, realized through Ptolemy II, may help the design process and reuse components in different phases. We will start with a simple continuous controller and step by step extend its behavior until we reach a description that captures important properties that arises in an actual implementation. In the final iteration, the model will, for example, include timing variations that are due to the fact that the controller executes as a task in an RTOS, and latencies due to the fact that the controller system is distributed. The point we make is that the refinements and component reuse are only possible due to the concepts of domains, MoCs, and responsible frameworks.

A picture of the Furuta pendulum [28] is shown in Figure 10. The pendulum consists of two moving parts, an arm that moves in the horizontal plane and a pendulum that moves in the vertical plane. The goal of this controller design is to swing up and stabilize the pendulum.

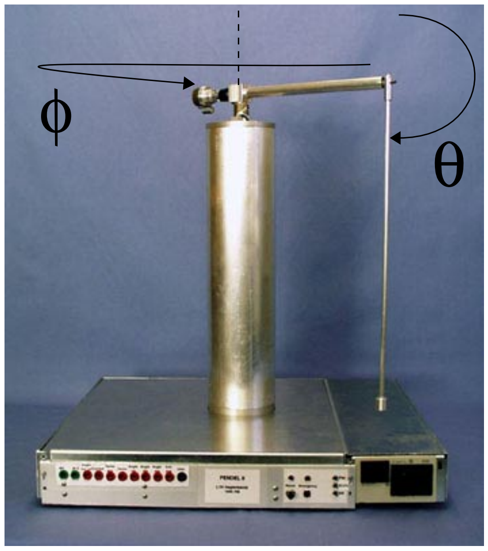

Fig. 7. The Furuta pendulum from the Department of Automatic Control at Lund University. The input signal is the torque to motor controlling the arm, and the output signals are the position and speed of the arm ($\theta, \dot{\theta}$) and the position and speed of the pendulum ($\phi, \dot{\phi}$).

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

1. continuous-time pendulum modeling;
2. continuous-time controller design for stabilization;
3. discrete-time controller design for stabilization with zero execution delay;

4. discrete-time modal controller design with three control modes – swing-up, catch, and stabilize;
5. discrete controller design with RTOS scheduling;
6. discrete controller design with network integrated sensors and actuators;
7. hardware-in-the-loop simulation with embedded implementation;
8. deployed realization.

Note that, in steps 1 to 6, we mainly enrich the design by adding new concerns and components to the system model. Steps 7 and 8 realize part or all systems physically. While the modeling and design within Ptolemy II has been achieved, systematically supporting hardware-in-the-loop simulation is still ongoing work.

The first step is to create the dynamic model for the inverted pendulum, as developed in [29]:

$$\begin{aligned}
\dot{x_1} =& x_2 \\
\dot{x_2} =& \frac{1}{d}\big(\alpha\beta + \alpha^2 \sin^3(x_1)\cos(x_1)x_4^2 - \gamma^2 x_2^2 \sin(x_1)\cos(x_1) \\
& + 2\alpha\gamma x_2 x_4 \sin(x_1)\cos^2(x_1) + \frac{\epsilon}{\alpha}\sin(x_1)(\alpha\beta + \alpha^2\sin^2(x_1))\big) \\
\dot{x_3} =& x_4 \\
\dot{x_4} =& \frac{1}{d}\big(-\alpha\gamma x_4^2 \sin(x_1)\cos^2(x_2) - \gamma\epsilon\sin(x_1)\cos(x_1) \\
& + \alpha\gamma x_2^2 \sin(x_1) - 2\alpha^2 x_2 x_4 \sin(x_1)\cos(x_1) + \alpha g u\big) \\
d =& \alpha\beta + \alpha^2\sin^2(x_1) - \gamma^2\cos^2(x_1)
\end{aligned} \tag{1}$$

where α, β, γ and g are constants, and x_i, $i = 1 \dots 4$ are the state variables representing the angle θ of the horizontal arm, the angle ϕ of the vertical arm, and their derivatives. The DifferentialSystem actor in Ptolemy II allow us the enter the ODE in the form of Equation 1. The angleConversion composite actor[3] restricts 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 linearized state feedback controller, as shown in Figure 8. An execution result is shown in Figure 9.

The next step is to discretize the continuous plants and calculate a discrete sample-data controller. As in many paper designs, the discrete controller is considered to have no delay, i.e. the actual execution time from sampling to actuation is neglected. 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 the designer's mind or informally on paper); implementing the three specifications by executable components (in this case, choosing components

[3]This is a *transparent* composite actor, which does not have a director but leverages the director on a level up (the CT Director, in this case).

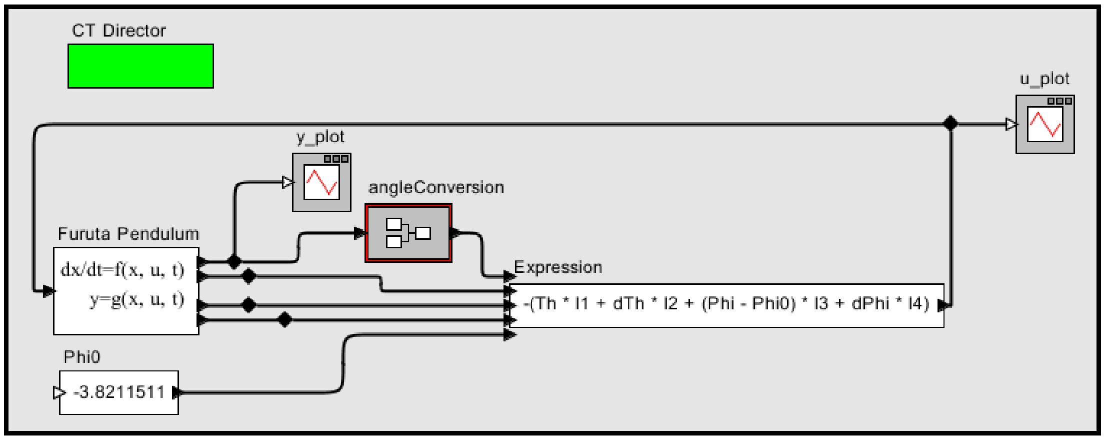

Fig. 8. A continuous-time pendulum controller.

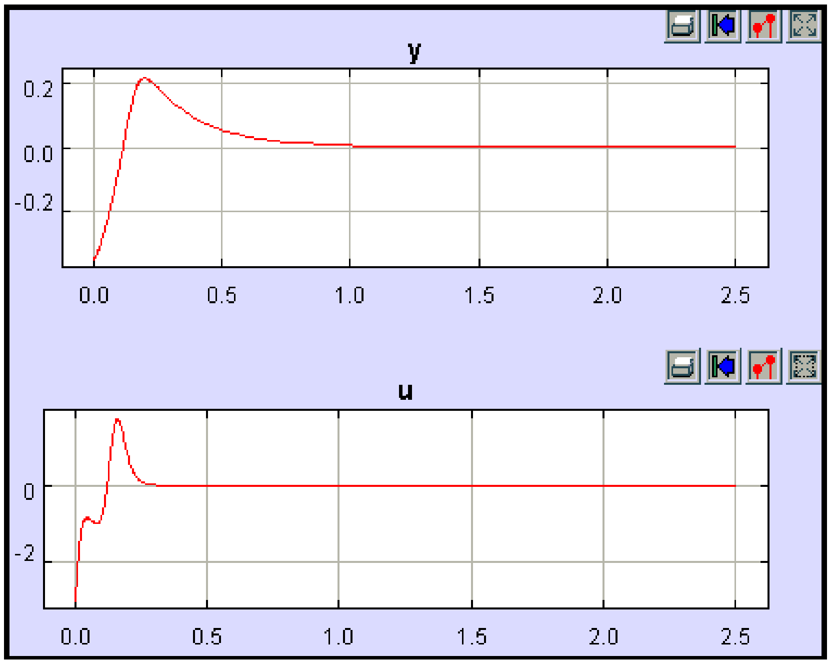

Fig. 9. The simulation results for a continuous-time stabilizing controller. Plot y shows the angle θ of the upper pendulum, and plot u shows the control signal.

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.

The Ptolemy II model at the end of this step is shown in Figure 10. 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 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

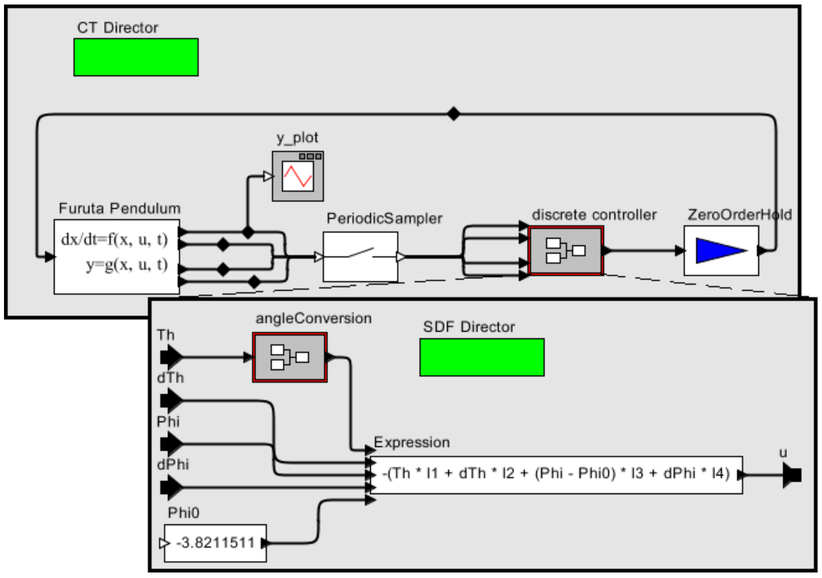

Fig. 10. A hierarchical model for the discrete-time stabilization controller.

on continuous-time signals (in Figure 8) and discrete data (in Figure 10), a property we call *domain polymorphism*.

The stabilizing controller only works when the pendulum already has an initial position close to the upward equilibrium. This is due to linearization. To swing up the pendulum from the initial downward position, we use a modal controller with three discrete states: swingup, catch, and stabilize.

To achieve this, a swingup controller and a catch controller are separately specified, implemented, and composed with the existing controller. Like the stabilization controller, the catch controller is a state feedback controller 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 [30], which injects the energy needed for the pendulum to move from the downward to the upward equilibrium, i.e. $\Delta E = 2mgl$ where m and l are the mass and length of the pendulum. In every sample step, the controller computes the current energy of the pendulum and produces a control output that increases (or decreases) the energy of the pendulum, so that it eventually reaches the top position with little or no extra energy (if the speed is too high the catch controller cannot take over). The three controllers are then composed through the FSM domain in Ptolemy II, as shown in Figure 11. This controller now replaces the discrete controller actor in Figure 10. Note that since the angleConversion actor 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

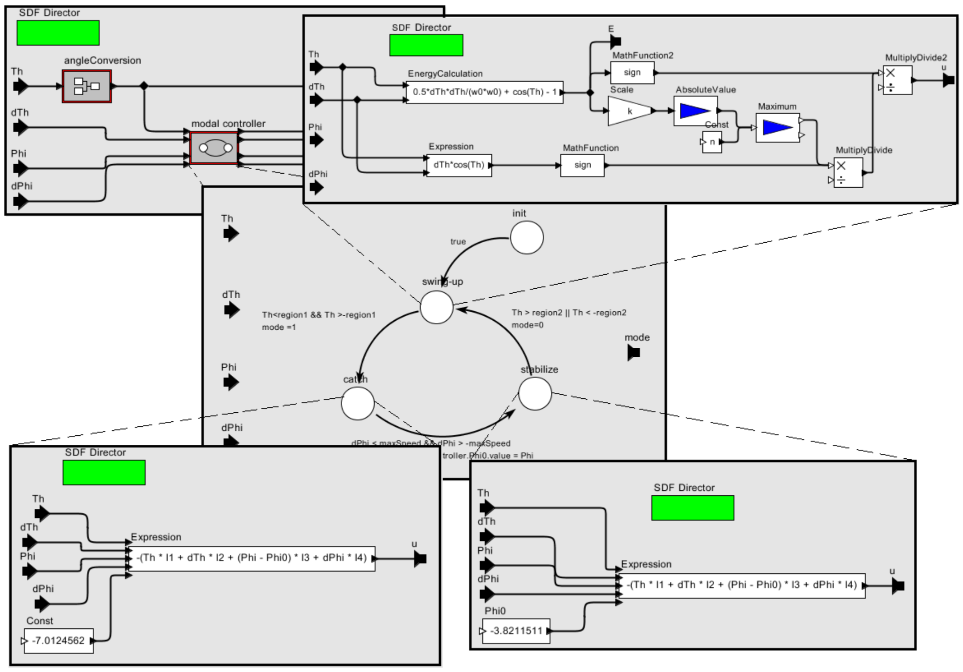

Fig. 11. A multi-mode, discrete-time controller for swinging up and stabilizing the pendulum. Implemented in a hierarchical and heterogeneous fashion.

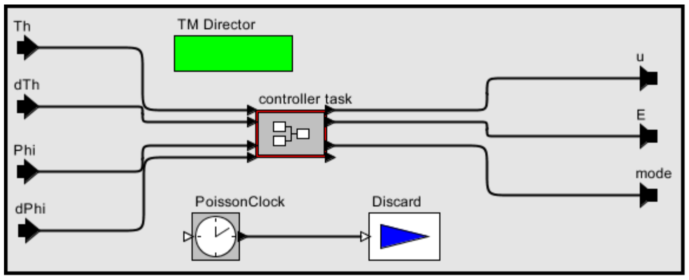

Fig. 12. Further refinement of the models includes timing and latency properties, that simulate that the controller is executing on a real CPU in a prioritized multitasking environment.

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 11 with the Discard actor that models other tasks in the system, as shown in Figure 12. The Discard actor, triggered by a Poisson clock, discards the input data value, but affects execution time of the controller. Figure 13 (a) and (b) show 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 controller parameters. For example, using another set of parameters, the controller is again able to swing up the pendulum (Figure 13 (c)), but it now needs almost five seconds and several attempts to swing up the pendulum, and

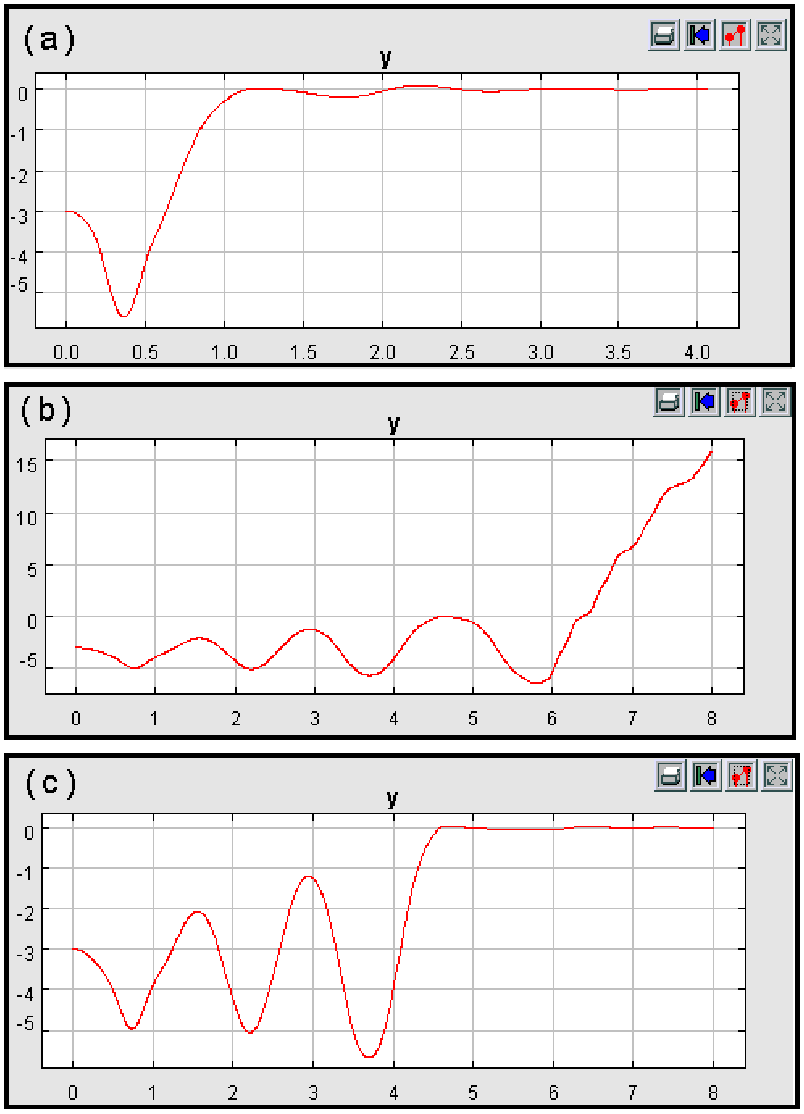

Fig. 13. Execution results with and without delay: (a) original parameters without execution delay, (b) original parameters with execution delay, (c) compensated parameters with execution delay.

the performance decreases due to the delays.

Next, we consider a distributed implementation of the system, where the controller is connected to the sensors and the actuator through a network. At the network level, all communication packets can be treated as events, which may occur at any point in time. So, we use the discrete-event domain to model the message passing between the actors that model the sensors, the actuator, and the controller. The sensors and the actuator are modeled as a pure time delays, 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 are modeled using a single “disturbance” node. Figure 14 shows the part of the final controller model where a CSMA/CD

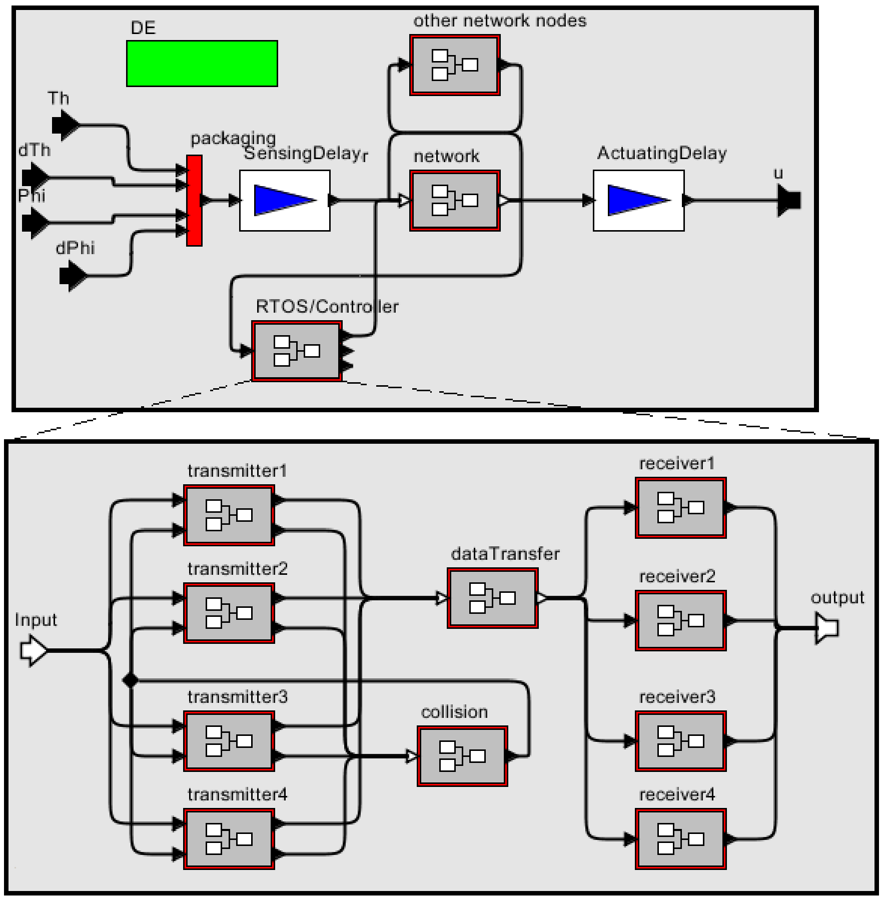

Fig. 14. The network part of a distributed controller model.

style media access protocol (e.g. Ethernet) is used. The transmitters try to send packets and listen for possible collisions. Only when no collision occurs during the entire sending period will a packet reach the receiver. 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 and delay variations to the control path. This again requires an exploration of control laws and mode transition conditions.

Hardware-in-the-loop simulation enables the real software and computer hardware to be integrated with simulated continuous dynamics of the plant. This is a particularly useful step for safety-critical systems where a failure in direct system deployment may cost too much. A real controller and a real network realize the embedded hardware and software, and replace the controller block at the top level of Figure 8. The dynamics of the pendulum, together with sampling and zero-order hold circuits, are still implemented in the Ptolemy II model. The actor-oriented modeling tool defines a clear boundary for such integration.

VI. Conclusion

This paper suggests an actor-oriented design methodology for developing complex control systems, and demonstrates the importance of responsible frameworks to enable hierarchical composition of heterogeneous models. Actor orientation localizes interactions among the components of a system. Hierarchies of models of computation effectively manage heterogeneity in the system, and precise reactions and responsible frameworks allow these models to be composable. This methodology is implemented through the Ptolemy II software environment. A pendulum swinging-up control system is used as an example to illustrate the step-by-step refinement of a design. Design concerns are gradually added to enrich the model and bring it closer to realization. Closed-loop control performance can be simulated and quickly fed back to designers at each step.

Acknowledgments

The authors would like to thank the entire Ptolemy team for inspiring discussion and collaborative implementation on the Ptolemy II software. In particular, John Reekie made significant contribution to an early version of this article, and Steve Neuendorffer and Yuhong Xiong have contributed key infrastructure to Ptolemy II that enabled this research.

References

[1] J. Lygeros, K. H. Johansson, S. N. Simic, J. Zhang, and S. Sastry, "Dynamical properties of hybrid automata," *IEEE Transactions on Automatic Control*, vol. 48, no. 1, 2003, ,to appear.

[2] 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, "Ptolemy II: Heterogeneous concurrent modeling and design in Java," EECS, University of California, Berkeley, Technical Memorandum UCB/ERL M01/12, March 2001.

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

[4] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language: User's Guide*. Addison-Wesley, 1999.

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

[6] 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 (WISP2001)*, Budapest, Hungary, May 2001.

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

[8] *Common Object Request Broker Architecture*. Object Management Group (OMG), July 1999.

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

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

[11] J. Paunicka, B. Mendel, and D. Corman, "The OCP – an open middleware solution for embedded systems," in *Proceedings of the American Control Conference(ACC'2001)*, Arlington, VA, May 2001, pp. 3345–3350.

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

[13] G. A. Agha, "Concurrent object-oriented programming," *Communications of the ACM*, vol. 33, no. 9, pp. 125–141, 1990.

[14] ——, *ACTORS: A Model of Concurrent Computation in Distributed Systems*, ser. The MIT Press Series in Artificial Intelligence. Cambridge, MA: MIT Press, 1986.

[15] D. B. Stewart, R. Volpe, and P. Khosla, “Design of dynamically reconfigurable real-time software using port-based objects,” *IEEE Trans. on Software Engineering*, vol. 23, no. 12, pp. 759–776, Dec 1997.

[16] N. Lynch, R. Segala, F. Vaandrager, and H. Weinberg, “Hybrid I/O automata,” in *Hybrid Systems III, LNCS 1066*, R. Alur, T. Henzinger, and E. Sontag, Eds. Springer-Verlag, 1996, pp. 496–510.

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

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

[19] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, “Ptolemy: A framework for simulating and prototyping heterogeneous systems,” *International Journal of Computer Simulation, special issue on “Simulation Software Development,”*, vol. 4, pp. 155–182, April 1994.

[20] 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,” *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, Jan. 2003.

[21] G. Kahn and D. B. MacQueen, “Coroutines and networks of parallel processes,” in *Proceedings of the IFIP Congress 77*, International Federation for Information Processing. Paris, France: North-Holland Publishing Company, 1977, pp. 993–998.

[22] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall, April 1985.

[23] E. A. Lee and T. M. Parks, “Dataflow process networks,” *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.

[24] E. A. Lee and D. G. Messerschmitt, “Synchronous data flow,” *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sept. 1987.

[25] J. Liu and E. A. Lee, “A component-based approach to modeling and simulating mixed-signal and hybrid systems,” *ACM Trans. on Modeling and Computer Simulation, special issue on Computer Automated Multi-Paradigm Modeling*, 2003, to appear.

[26] ——, “Timed multitasking for real-time embedded software,” *IEEE Control Systems*, vol. 23, no. 1, pp. 65–75, 2003.

[27] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman, “On the formal semantics of Statecharts,” in *Proceedings of the Symposium on Logic in Computer Science*, June 1987, pp. 54–64.

[28] K. Furuta, M. Yamakita, S. Kobayashi, and M. Nishimura., “A new inverted pendulum apparatus for eduction,” in *IFAC Symposium on Advances in Control Education*, Boston, MA, 1994, pp. 191–196.

[29] M. Gäfvert, “Derivation of Furuta pendulum dynamics,” Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, Tech. Rep., 1998.

[30] K. J. Åström and K. Furuta, “Swinging up a pendulum by energy control,” *Automatica*, vol. 36, pp. 278–285, 2000.