Proceedings of the 2004 Winter Simulation Conference
R. G. Ingalls, M. D. Rossetti, J. S. Smith, and B. A. Peters, eds.

MODELING REAL-WORLD CONTROL SYSTEMS: BEYOND HYBRID SYSTEMS

Stephen Neuendorffer

EECS Department
University of California at Berkeley
Berkeley, CA 94720, U.S.A.

ABSTRACT

Hybrid system modeling refers to the construction of system models combining both continuous and discrete dynamics. These models can greatly reduce the complexity of a physical system model by abstracting some of the continuous dynamics of the system into discrete dynamics. Hybrid system models are also useful for describing the interaction between physical processes and computational processes, such as in a digital feedback control system. Unfortunately, hybrid system models poorly capture common software architecture design patterns, such as threads, mobile code, safety, and hardware interfaces. Dealing effectively with these practical software issues is crucial when designing real-world systems. This paper presents a model of a complex control system that combines continuous-state physical system models with rich discrete-state software models in a disciplined fashion. We show how expressive modeling using multiple semantics can be used to address the design difficulties in such a system.

1 Introduction

Hybrid system formalisms (Lygeros, Tomlin, and Sastry) are often used to simply represent and describe the behavior of a physical system that cannot be easily described using differential equations alone. Less commonly, hybrid system formalisms are used to describe the interaction between discrete computation systems and the physical world. This interaction often arises in the form of *embedded systems*, such as digital control systems (Liu, Liu, Koo, Sinopoli, Sastry, and Lee 1999, Eker, Fong, Janneck, and Liu 2001), high-performance data acquisition systems (Ludvig, McCarthy, Neuendorffer, and Sachs 2002), and heterogeneous electronic systems containing analog and digital components (Liu 1998). In these cases, the interaction between the discrete and analog portions of a system are tightly coupled and crucial to the proper behavior of the system. We would like to model the behavior of such systems abstractly as hybrid systems in order to quickly design the function of discrete computation.

Unfortunately, although hybrid system formalisms provide a framework for describing the interaction between software and the physical world, hybrid systems are not a good model for designing embedded software. It is not generally practical to explicitly model the computational states of embedded software through the discrete states of a hybrid system model because of the large number of explicit states required. Furthermore, such a model bears little resemblance to efficient executable code, making implementation synthesis difficult.

We have approached this problem by constructing a design environment, called Ptolemy II (Davis et al. 2001), that is capable of modeling both the hybrid dynamics of physical systems and complex software architectures. Ptolemy II emphasizes the disciplined construction of *hierarchically heterogeneous* system models using multiple modeling semantics, called *models of computation* (Lee 2002). In Ptolemy II, each model of computation is represented by a *director* that gives modeling semantics to a particular level of hierarchy in the model. Hierarchical heterogeneity (Eker et al. 2003) allows discrete-state and continuous-state models to be constructed from primitive components and be robustly composed to build hybrid system models. These models can be further composed with models of embedded software constructed using semantics appropriate for software design. Since every model exposes an opaque component interface that completely describes all interaction with other components, complex models can be constructed from smaller models without overwhelming a system architect.

We call this style of component-based modeling *actor-oriented design*. One advantage of actor-oriented modeling is abstraction: systems can be represented at high levels of abstraction, facilitating rapid creation of executable simulation models. The functional behavior and current state of a component are encapsulated inside the component. Timing and concurrency can be represented directly as part of an actor-oriented model, enabling detailed simulation of software systems interacting with the physical world. Furthermore, since actors have a well-defined interface, the model of a hybrid embedded system can often be easily partitioned into a software model and an environment model, enabling hardware-in-the loop simulation and automatic implementation synthesis. Given correct, detailed models of a physical system, actor-oriented modeling provides a correct-by-construction path to the construction of embedded software. This paper illustrates these design techniques using a model of an autonomous vehicle controller for the Caltech Multi-vehicle Wireless Testbed (Cremean et al. 2002) as an extended example.

2 Caltech Multi-Vehicle Wireless Testbed

Murray *et al.* at Caltech have developed a platform for experimenting with coordinated control of autonomous vehicles, called the Multi-Vehicle Wireless Testbed (MVWT) (Cremean et al. 2002). The platform consists of a number of ground vehicles operating in a controlled environment. Propulsion for each vehicle is provided by a pair of ducted fans mounted on top of the vehicle. By applying the same force to each fan, the vehicle will move forward in a straight line, while applying a different force to each fan causes the vehicle to turn. An embedded computer controls the fans through off-the-shelf motor controllers connected to an RS-232 interface. The vehicles slide on three industrial casters, allowing them to slide sideways while turning. The operating environment of the vehicles includes a video camera-based localization system (the *Lab Positioning System*), which broadcasts location and orientation information for each vehicle over 802.11b wireless ethernet using UDP datagrams. Because of the embedded, highly mobile nature of the vehicles, control commands such as way points are entered from a separate base station computer and transmitted to individual vehicles.

From a control-oriented viewpoint, the continuous-time dynamics of a Caltech vehicle can be modeled by the equations (from (Cremean et al. 2002)):

$$m\,\ddot{x}(t) = -\eta\,\dot{x}(t) + [F_s(t) + F_p(t)]\cos\theta(t)$$

$$m\,\ddot{y}(t) = -\eta\,\dot{y}(t) + [F_s(t) + F_p(t)]\sin\theta(t)$$

$$J\,\ddot{\theta}(t) = -\psi\,\dot{\theta}(t) + [F_s(t) - F_p(t)]\,r$$

where friction is assumed to be proportional to velocity. $F_s(t)$ and $F_p(t)$ are inputs to the system corresponding to the forces applied to the starboard and port fans respectively. The system state $\sigma(t) = [x(t), \dot{x}(t), y(t), \dot{y}(t), \theta(t), \dot{\theta}(t)]^T$ is available to a control algorithm through the localization system. The vehicle dynamics are similar to a two dimensional approximation of winged aircraft dynamics (Evans, Inalhan, Jang, Teo, and Tomlin 2001), making the testbed useful for experimenting with aircraft control systems. The Caltech group has implemented LQR-based state-feedback digital control of the above dynamics capable of tracking smooth trajectories.

In the physical system, there are several hardware limitations that serve to complicate the design of the control system. For instance, in the physical system, F_s and F_p are limited to a maximum of approximately 5 Newtons, and cannot operate in reverse. The fans are driven by discrete-input motor controllers, resulting in quantization of the forces that can actually be applied to the vehicle. The localization system is only capable of capturing 60 frames of video per second, limiting the availability of location estimates. Lastly, the computational system is distributed (between the localization system, the base station and various vehicle controllers) and communicates over a shared media with the potential to lose data.

While these issues are partially addressed through robust LQR control system design techniques, simulation is still a crucial step in the design of such a system. In particular, some system requirements must be specified explicitly at the software level. For instance, one design requirement of the system is that the base station must be able to record telemetry from the vehicle and dynamically reconfigure the executing controller. A safety requirement may state that if the communication network fails or if the vehicle begins to spin out of control, that the controller should power down automatically. Simulation allows for these scenarios to be modeled and appropriate action integrated into the software model. Effectively modeling such behaviors, which might be implemented using dynamic loading of mobile code or software reconfiguration, requires software architecture models that are fundamentally different from models of the physical system. However, these models must be able to interact with physical system models in a simulation environment.

3 Basic Control Model

A Ptolemy II model of the physical dynamics of a single vehicle is shown in Figure 1. This model is constructed in the style of Simulink, a commercial tool produced by The Mathworks Inc. The semantics of component interaction are designed to support numerical integration algorithms, implemented by the `Integrator` component. The signals communicated between components are interpreted as functions of time that are solutions to a set of Ordinary Differential Equations. We call this style of modeling, with continuous-time signals and numerical integration, the continuous-time (CT) model of computation.

In this model, the inputs are not taken as continuous functions, but are instead assumed to be discrete-event

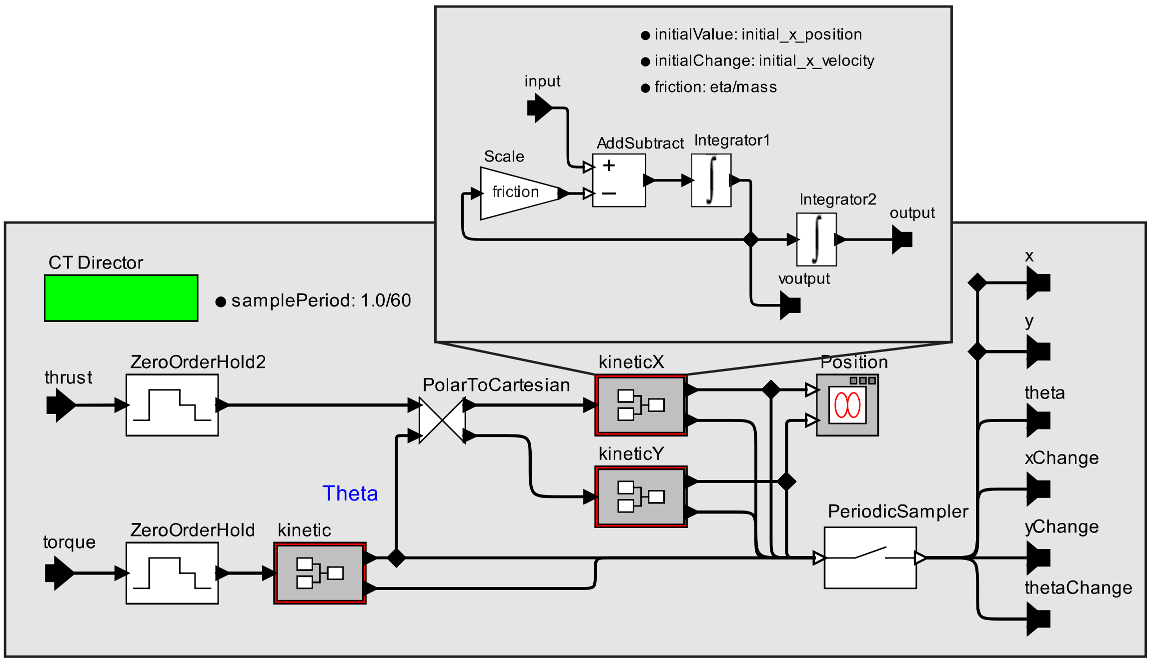

Figure 1: A model of the continuous dynamics of a single vehicle. The hierarchical `kinetic` and `kineticY` models are similar to `kineticX`. Each connection in this model represents a continuous-time signal with a value defined at all points in time. Simulation approximates the values of these signals at desired points in time.

signals. Discrete-event signals, unlike continuous-time signals, are assumed to take values only at a countable number of points in time. At all other points in time, a discrete-event signal has no value and is said to be *absent*. We call the style of modeling, with chronological processing events in discrete-event signals, the discrete-event (DE) model of computation. The `Zero-Order Hold` components convert from discrete-event signals into continuous-time signals suitable for integration. Similarly, the `PeriodicSampler` component produces a discrete-event signal, which happens to consist of events evenly spaced in time, from the continuous-time signal. In this model, the `PeriodicSampler` models the fact that only sample values of the continuous-time dynamics are available to the vehicle controller.

The interaction between the vehicle model above, and the controller is shown in Figure 2. This model includes a detailed model of the data format between the plant and the controller. The localization system broadcasts a UDP datagram containing the encoded state of the vehicle, approximately 60 times a second. This communication is modeled by an event consisting of a 56-byte array sent from the vehicle model to the controller model. In response to each network packet, the control computer executes the control algorithm and eventually sends a three byte serial sequence to change the speed of the fans. The serial communication is modeled by a separate event sent from the controller. Conversions to and from arrays of bytes are modeled by `Extract Forces` and `Construct Localization Packet`, which are not shown in detail.

Unfortunately, from the point of view of accurate simulation, we have no idea how long the controller computation will actually take in the final system. In this model, the controller is idealized and generates its output event in zero time. The model includes an explicit model of the computation and communication delay, given by the `TimedDelay` component. Here the delay is assumed to be constant, but it is trivial to substitute a stochastic delay, perhaps allowing for the possibility of dropped packets.

The model of the controller itself is shown in Figure 3. This controller is based on an LQR controller design by Murray's group for tracking circular trajectories. The trajectories are generated in polar form according to parameters specified in the model. Note that input to the `Circular Trajectory Controller` is a structured record datatype containing six fields, one for each state variable of the physical system. The state is converted to another record containing the state in polar form and the control law is computed in polar space. This controller uses an idealized model of the forces produced by each fan (implemented by `Force Map`), which in a real system would be replaced by calibrated lookup tables. A simulation plot is shown in Figure 10.

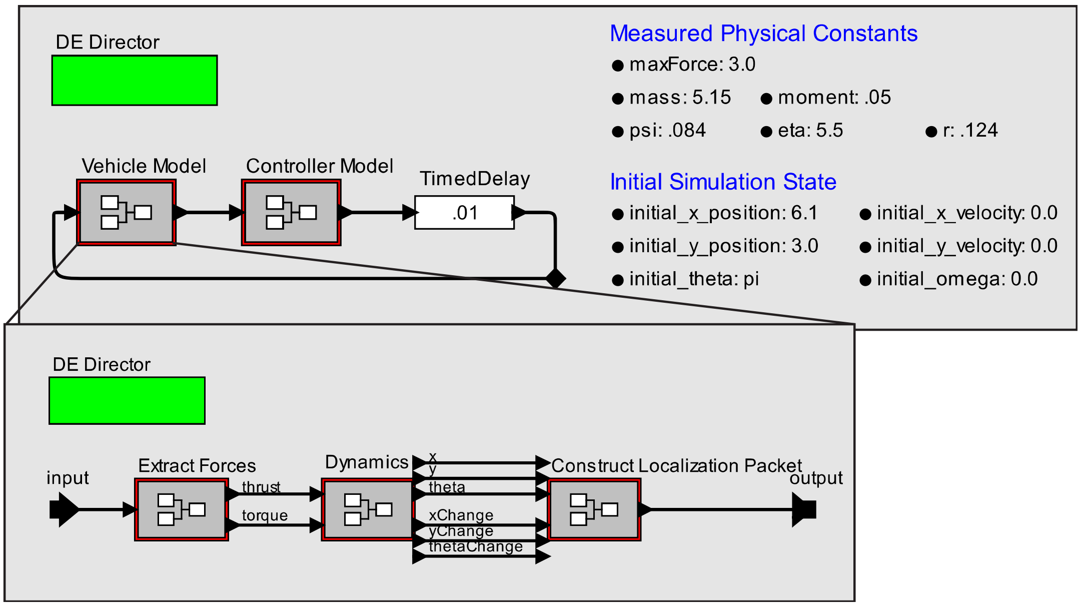

Figure 2: The toplevel simulation model, showing the interaction between the model of vehicle dynamics and the model of the controller. Each connection in these model represents a discrete-event signal whose value is only defined at discrete points in time.

The controller model is constructed as a Synchronous Dataflow (SDF) model. (Lee and Messerschmitt 1987) In this model, the signals between components are *untimed sequences* of values. Each component processes these sequences in order at fixed relative data rates, allowing the execution of individual components to be statically ordered in a fixed schedule. Because of the nice scheduling properties and the ability to synthesize efficient implementations (Bhattacharyya, Murthy, and Lee 1996), Synchronous Dataflow is a good model of computation for dataflow-oriented embedded software.

4 Implementation

The model in previous section is a model that is constructed primarily to allow simulation. Some aspects of the intended system have been modeled explicitly, such as the format of information received from the localization system. On the other hand, some aspects of the system have been abstracted, such at the communication between the localization system and the controller. The model represents this communication as an instantaneous event, while the actual communication layer incurs some random (possibly infinite) delay. The model of the vehicle is, itself an abstract representation of the vehicle and the localization system. These models cannot be viewed as a program, i.e. a source for synthesis, without additional information, such as a communication protocol or a 3D CAD model of the physical vehicle.

The controller on the other hand, is a concrete model. Given appropriate inputs and outputs, the controller is in a form which directly corresponds to a software architecture for implementing the controller algorithm. This architecture can be automatically generated in a relatively straightforward mapping from the original. However, in order to perform the synthesis procedure, this the concrete, synthesizeable portion of the simulation model (corresponding to embedded software) is partitioned from the abstract portion. The result of partitioning the above system is shown in Figure 6. Note that the communication channels have been replaced with `Datagram`, and `SerialComm` components encapsulating the UDP and RS-232 communication interfaces.

Note that Figure 6 includes the entire partitioned model, including the abstract portion corresponding to the vehicle dynamics. While this portion is not useful for implementation synthesis, it can be used for distributed simulation. By executing the model of the vehicle on one computer and a model of the controller on another computer, more accurate simulation of the system behavior can be performed. In particular, such a simulation includes actual properties of the communication protocol, rather than the approximate model included earlier. Other combinations are also possible, resulting in various forms of hardware-in-the-loop simulation (Sanvido 2002). For instance, the controller model can be executed in the actual system, taking the place of an embedded controller. This structure allows us to test that communication protocols and vehicle dynamics have been

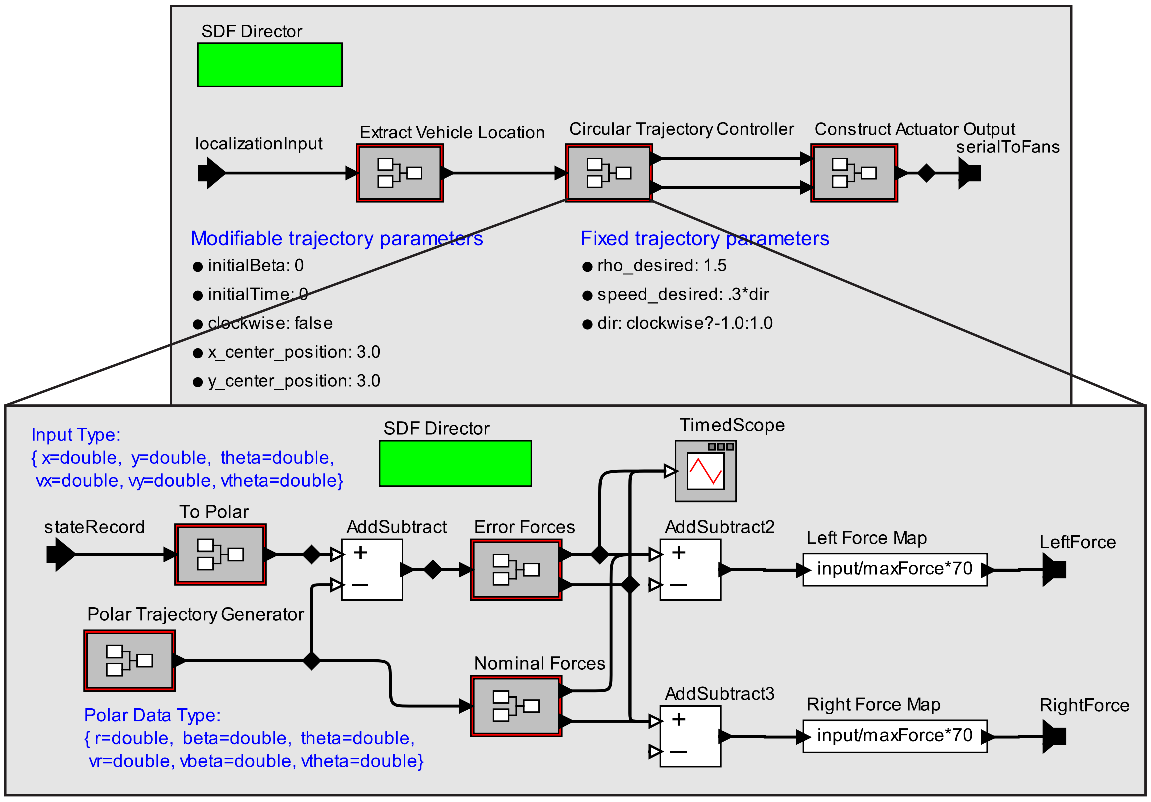

Figure 3: Model of the vehicle controller. `Extract Vehicle Location` decodes localization information from an array of bytes into a record of values and `Construct Actuator Output` encodes the control output into an array of bytes. The interesting part of the controller is implemented by `Circular Trajectory Controller`, and is shown expanded below. Each connection in this model represents an untimed sequence of values which are processed in order.

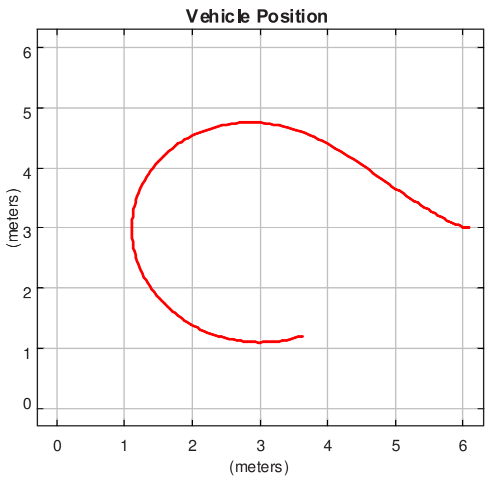

Figure 4: A simulation plot of the position of a vehicle, tracking a counter-clockwise circular trajectory around the point (3,3).

```
time = 0.0000, {85ub, 70ub, 0ub}
time = 0.0167, {85ub, 70ub, 0ub}
time = 0.0334, {85ub, 70ub, 0ub}
time = 0.0501, {85ub, 70ub, 0ub}
time = 0.0668, {85ub, 70ub, 0ub}
time = 0.0835, {85ub, 70ub, 0ub}
time = 0.1002, {85ub, 70ub, 0ub}
time = 0.1169, {85ub, 70ub, 0ub}
time = 0.1336, {85ub, 70ub, 34ub}
time = 0.1503, {85ub, 70ub, 70ub}
time = 0.1670, {85ub, 70ub, 70ub}
```

Figure 5: A partial trace of the discrete-event signal output from the controller during simulation. The value of each event is an array of three unsigned byte values between zero and 255. The array consists of a dummy start byte (85), followed by values between zero and 70 for each fan, where a value of 70 corresponds to maximum available thrust.

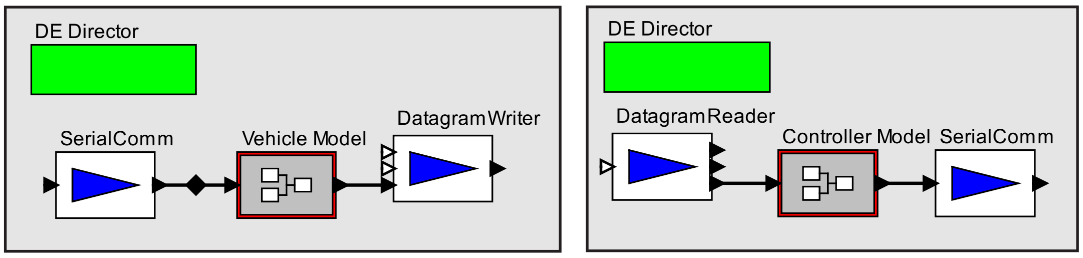

Figure 6: A partitioned version of the simulation model, where event communication has been replaced with communication interfaces. The `VehicleModel` and `Controller` models are as before. In this model, `SerialComm` encapsulates an RS-232 serial interface and the `DatagramReader` and `DatagramWriter` encapsulate event-driven communication using UDP datagrams.

modeled in sufficient detail. Alternatively, the vehicle dynamics model can be executed with code generated from the controller model, to test that the implementation was generated correctly.

5 Improving the System Model

The model presented above includes many details that are abstracted by the differential equation dynamics. However, from an embedded software perspective, the model is still very minimal. It does not model how the system is initialized, for instance, or how the system recovers from errors. This information must either be specified as part of code generation, perhaps by specifying a target platform that provides initialization and reset capabilities, or it must be specified through a more detailed system model. In order to show how these might be represented in a more detailed model, we concentrate on the interaction between the control algorithm and the base station computer.

The first improvement we consider is the ability to trigger mode switches from the base station. This is modeled by augmenting the model of the controller with a finite state machine to control reconfiguration, as shown in Figure 7. In each mode, the modal controller behaves as the original controller, which follows circular trajectory given by a set of parameters. In response to switch events sent by the base station over a separate UDP datagram port, the state machine enters an intermediate *switching state*. The state machine waits in the switching state until the position of the vehicle is reasonably close to the new trajectory, at which time the state machine automatically transitions to a new state, reconfiguring the control algorithm to follow the new trajectory. The guard leaving the switching state must be designed so that the new trajectory can be followed without saturating the available control inputs. In the new controller state the vehicle follows a circular trajectory with the new parameters. In this case, the controller only switched between two fixed trajectories, in order to emphasize the presence of switching states. In general, the parameters of the new trajectory and the switching guard could be received as part of the request for a trajectory change.

The second improvement that we deal with is the ability of the base station to dynamically update and modify the control algorithm remotely. This is modeled using a `MobileModel`, as shown in Figure 8. This component does not have behavior of its own, but simply encapsulates other components received on its bottom input port. In this case, the mobile model receives a description of the component over a CORBA-based publish and subscribe network, encapsulated by the `PushConsumer` and `PushSupplier` components. Essentially, the controller publishes a event service which the base station computer subscribes to, allowing it to push a new component description to the controller. Although it was not represented here, switching guards are often important when updating components, in order to avoid control transients.

A final improvement to the model addresses the need for automatic shutdown of the system in case of network failure. A modified controller model is shown in figure 9. This model moves from a purely event-driven style of execution, where the controller was driven by the arrival of a network packet, to a more time-driven Giotto model of computation (Henzinger, Horowitz, and Kirsch 2001, Henzinger, Kirsch, Sanvido, and Pree 2003). A Giotto model presents an abstract view of time-triggered tasks executing in a real-time operating system with deterministic communication. Each signal in the Giotto model is a special case of discrete-event signal where events occur periodically in time. Like Synchronous Dataflow, Giotto is a model of computation that can be synthesized into efficient embedded software.

In the model, the state of input channels is updated based on incoming events from the localization system and the time of the last event arrival on the input port to the Giotto model is provided explicitly. The controller task is triggered at a fixed rate of 100 Hz regardless of when

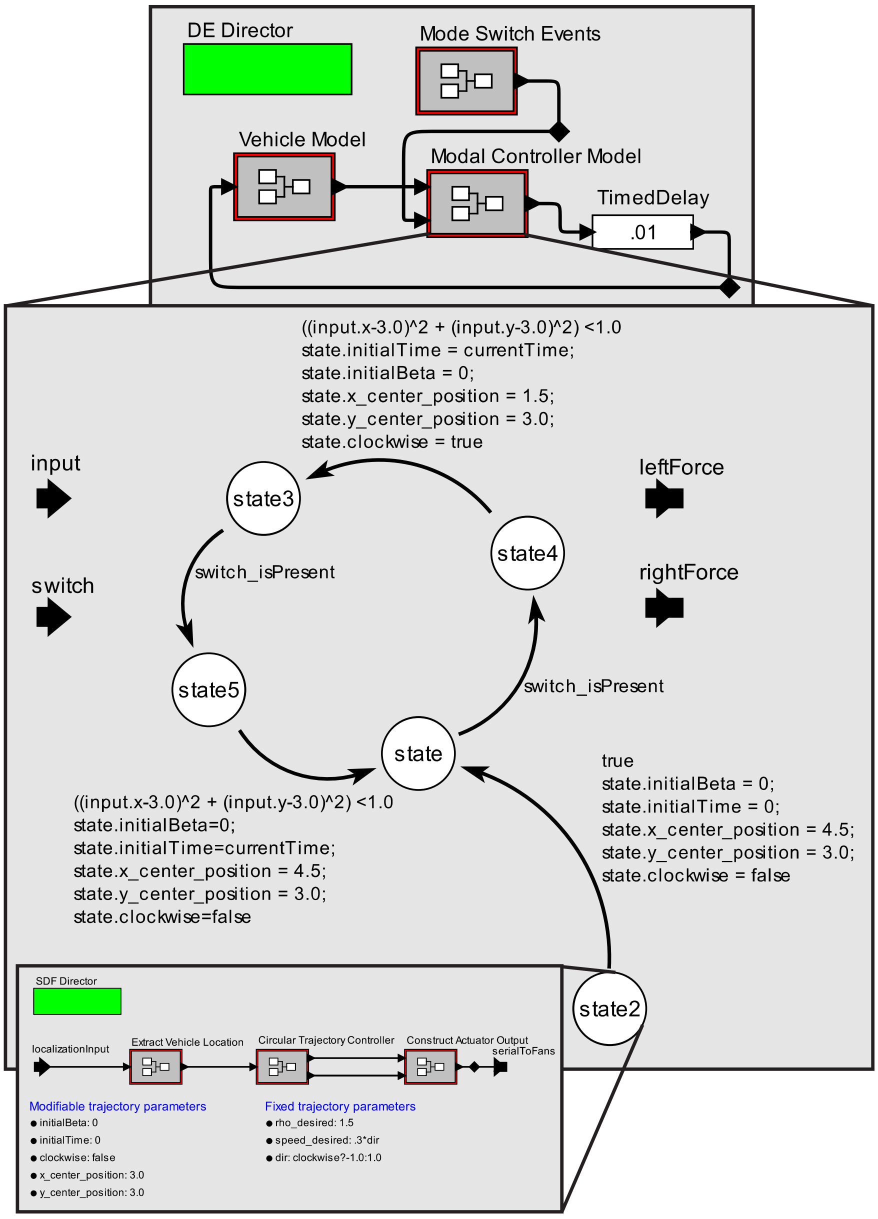

Figure 7: A model of a modal controller. The base station computer can send an event over the network, triggering a state transition and reconfiguration of the controller parameters.

input events arrive and always sees the newest available localization input. This architecture does not appreciably increase the latency of the controller, but ensures that the control algorithm is regularly executed, even in the absence of fresh localization data. Detection of the network failure is actually performed by the `SafetyShutdown` component. This component compares the current time with the time of the last localization event to determine whether or not a network failure has occurred. The `CurrentTime` component gives the current simulation time, as determined by the toplevel discrete-event driven model. In synthesized embedded software using a real-time operating system, the current time would be implemented using a real-time system clock. If a network failure is detected, feedback in the model forces the software to be reinitialized before the controller will be active again.

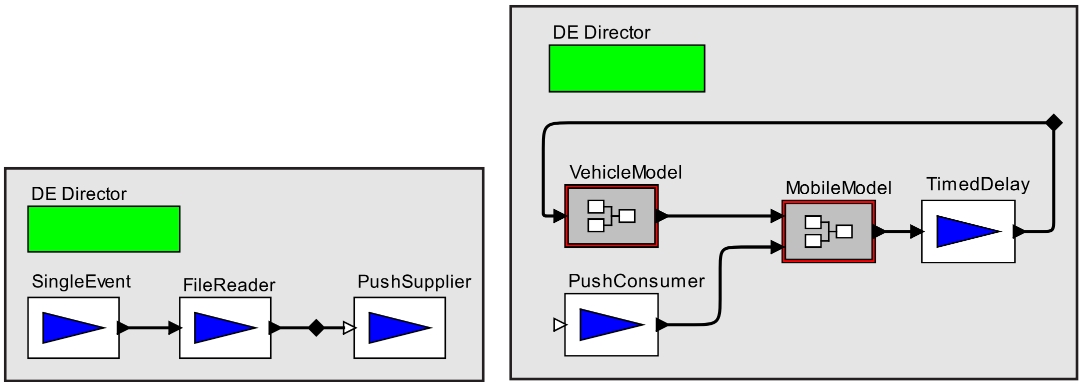

Figure 8: A model of the interaction between the base station computer (on the left) and the control system model on the right. The base station can remotely update the controller being executed. In this model, PushSupplier and PushConsumer encapsulate event-driven communication over a publish and subscribe network interface.

6 Conclusion

This paper has presented a sequence of Ptolemy II models illustrating techniques for modeling the behavior of embedded control systems. Part of the complexity in such systems can be handled by building heterogeneous models with multiple execution semantics. By starting with an abstract model that is close to a control engineer's conceptualization of the system, we have engineered more complex behaviors by leveraging more complex modeling idioms that do not fit into an analytical model of the control system. Ultimately, the model is designed so that control software (or hardware) can be automatically synthesized from the model after sufficient scenario-based simulation. This modeling approach combines idealized and concrete models of the system, and provides a path to gain understanding of idealized portions of the model through hardware-in-the-loop simulation.

REFERENCES

Bhattacharyya, S. S., P. K. Murthy, and E. A. Lee. 1996. *Software synthesis from dataflow graphs*. Kluwer.

Cremean, L. et al. 2002, December. The Caltech multi-vehicle wireless testbed. In *Proceedings of the Conference on Decision and Control (CDC)*: IEEE.

Davis, J. et al. 2001, March. Ptolemy II - Heterogeneous concurrent modeling and design in Java. Memo M01/12, UCB/ERL, EECS UC Berkeley, CA 94720.

Eker, J. et al. 2003, January. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE* 91 (1).

Eker, J., C. Fong, J. W. Janneck, and J. Liu. 2001, November. Design and simulation of heterogeneous control systems using Ptolemy II. In *IFAC Conference on New Technologies for Computer Control*. Hong-Kong, China.

Evans, J., G. Inalhan, J. Jang, R. Teo, and C. Tomlin. 2001, October. Dragonfly: A versatile UAV platform for the advancement of aircraft navigation and control. In *Proc. of the 20th Digital Avionics Systems Conference*. IEEE.

Henzinger, T. A., B. Horowitz, and C. M. Kirsch. 2001. Giotto: a time-triggered language for embedded programming. In *Proceedings of EMSOFT 01: Embedded Software*, Number 2211 in Lecture Notes in Computer Science, 166–184. Springer-Verlag.

Henzinger, T. A., C. M. Kirsch, M. A. Sanvido, and W. Pree. 2003. From control models to real-time code using Giotto. *IEEE Control Systems Magazine* 23 (1): 50–64.

Lee, E. A. 2002. Embedded software. *Advances in Computers* 56.

Lee, E. A., and D. G. Messerschmitt. 1987, September. Synchronous Data Flow. *Proceedings of the IEEE* 75 (9): 55–64.

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

Liu, J., X. Liu, T. J. Koo, B. Sinopoli, S. Sastry, and E. A. Lee. 1999, December. A hierarchical hybrid system model and its simulation. In *38th IEEE conference on Decision and Control, Phoenix, AZ.*

Ludvig, J., J. McCarthy, S. Neuendorffer, and S. R. Sachs. 2002, November. Reprogrammable platforms for high-speed data acquisition. *Journal of Design Automation for Embedded Systems* 7 (4): 341–364.

Lygeros, J., C. Tomlin, and S. Sastry. The art of hybrid systems. Unpublished manuscript, See also the *Conference on Hybrid Systems : Computation and Control.*

Sanvido, M. A. A. 2002, March. *Hardware-in-the-loop simulation framework*. Ph. D. thesis, ETH Zurich.

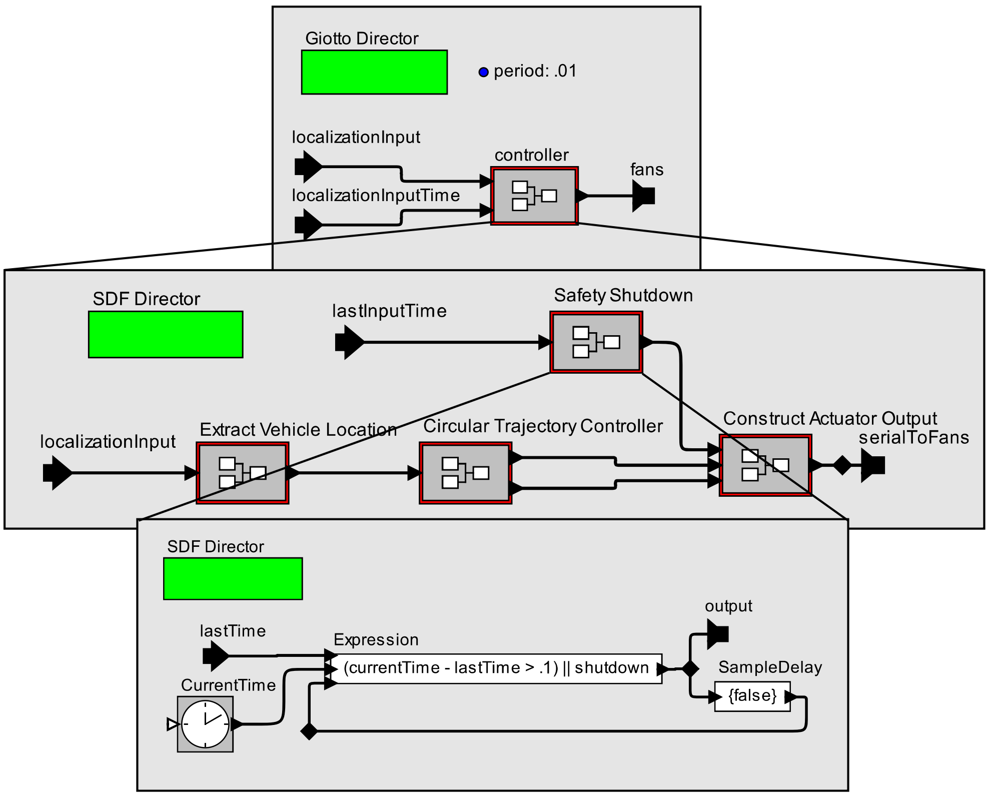

Figure 9: A modified controller that disables the vehicle if network failure is detected.

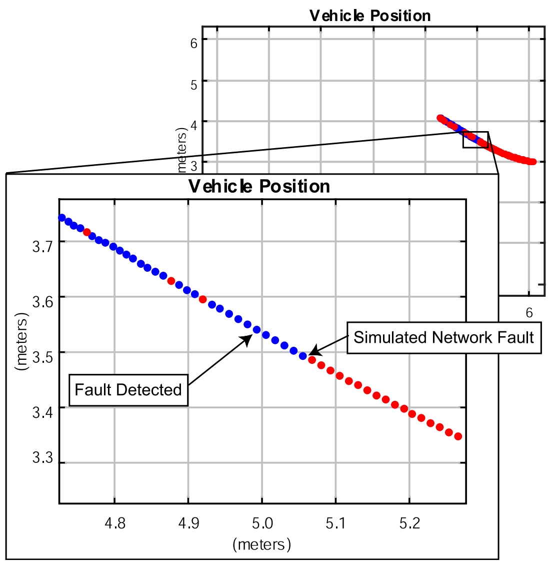

Figure 10: A plot of position estimates of a vehicle for a scenario with network failure. Received localization events are plotted in red, while missed events are plotted in blue. After .1 seconds, corresponding to 6 missed localization events, the vehicle controller assumes a network failure and disables the fans.