Chapter 4 from: C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng "Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains)," Technical Memorandum UCB/ERL M04/17, University of California, Berkeley, CA USA 94720, June 24, 2004.

4

FSM Domain

Authors: *Xiaojun Liu*
Edward A. Lee
Haiyang Zheng

4.1 Introduction

Finite state machines (FSMs) have been used extensively in designing sequential control logic. There are two major reasons behind their use. First, FSMs are a very intuitive way to capture control logic and make it easier to communicate a design. Second, FSMs have been the subject of a long history of research work. Many formal analysis and verification methods have been developed for them.

In their simple flat form, FSM models have a key weakness: the number of states in an FSM model can get quite large even for a moderately complex system. Such models quickly become chaotic and incomprehensible when one tries to model a system having many concurrent activities. The problem can be solved by introducing hierarchical organization into FSM models and using them in combination with concurrency models. David Harel first used this approach when he introduced the *Statecharts* formalism [50].

The Statecharts formalism extends the conventional FSM model in three aspects: hierarchical decomposition of states, concurrent composition of FSMs in a synchronous-reactive fashion, and a broadcast communication mechanism between concurrent components. While how these extensions fit together was not completely specified in [50], Harel's work stimulated a lot of interest in the approach. Consequently, there is a proliferation of variants of the Statecharts formalism [12], each proposing a different way to make the extensions fit into a monolithic model. Unfortunately, in all these variants FSM is combined with a particular concurrency model. The applicability of the resulting models is often limited.

Based on the Ptolemy philosophy of hierarchical composition of heterogeneous models of computation, the **charts*[1] formalism [45] allows embedding hierarchical FSMs within a variety of concurrency models. If tight synchronization is possible and desirable, then FSMs can be composed by the synchronous-reactive model. If the system has a global notion of time and components communicate

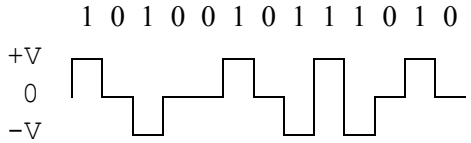
by time-stamped events, then FSMs can be composed by the discrete-event model. The rest of this chapter focuses on how the FSM domain in Ptolemy II supports the *charts formalism.

4.2 Building FSMs in Vergil

An FSM model is contained by an instance of FSMActor. The FSM model reacts to inputs to the FSM actor by making state transitions. Actions such as sending tokens to the output ports of the FSM actor can be associated with state transitions. In this section, we show how to construct and run a model with an FSM actor in Vergil.

4.2.1 Alternate Mark Inversion Coder

Alternate Mark Inversion (AMI) is a simple digital transmission technique that encodes a bit stream on a signal line as shown below:



The 0 bits are transmitted with voltage zero. The 1 bits are transmitted alternately with positive and negative voltages. On average, the resulting waveform will have no DC component.

We can model an AMI coder with a two-state FSM shown in figure 4.2. To construct a Ptolemy II model containing this coder, follow these steps:

1. Start Vergil, open a graph editor by selecting File -> New -> Graph Editor.
2. From MoreLibraries/Automata in the palette on the left, drag an FSM actor to the graph. Rename the FSM actor AMICoder.
3. Right click on AMICoder, select Configure Ports. Add an input port with name *in* and an output port with name *out* to AMICoder.
4. Right click on AMICoder, select Look Inside. This will open an FSM editor for AMICoder. Note that the ports of AMICoder are placed at the upper left corner of the graph panel.
5. From the palette on the left, drag a state to the graph, rename it Positive. Drag another state to the graph, rename it Negative.
6. Control-drag from the Positive state to the Negative state to create a transition.
7. Double click on the transition. This will bring up the dialog box shown in figure 4.1 for editing the parameters of the transition.
8. Set guardExpression to `in == 1`, and outputActions to `out = 1`.
9. Create a transition from the Positive state back to itself with guard expression `in == 0` and output action `out = 0`.

1. Pronounced “starcharts.” The star represents a wildcard that can be interpreted as matching multiple concurrency models.

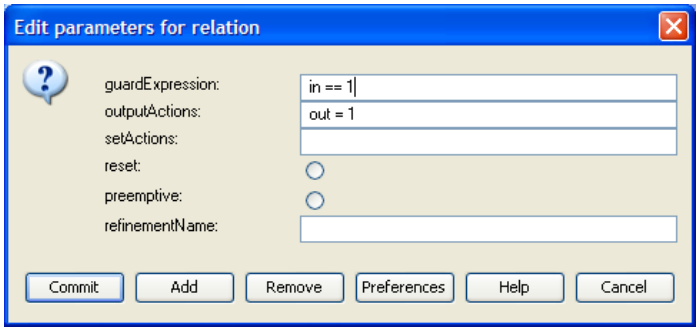


FIGURE 4.1. The dialog box for editing parameters of a transition.

10. Create a transition from the Negative state back to itself with guard expression `in == 0` and output action `out = 0`.
11. Create a transition from the Negative state to the Positive state with guard expression `in == 1` and output action `out = -1`.
12. Right click on the background of the graph panel. Select Configure from the context menu. This will bring up the dialog box for editing parameters of AMICoder. Set initialStateName to `Positive`.
13. The construction of AMICoder is complete. It will look like what is shown in figure 4.2.
14. Return to the graph editor opened in step 1.
15. Drag a Pulse actor (from Actors/Sources/SequenceSources), a SequencePlotter (from Actors/Sinks/SequenceSinks), and an SDF director (from Directors) to the graph.
16. Connect the actors as shown in figure 4.3.

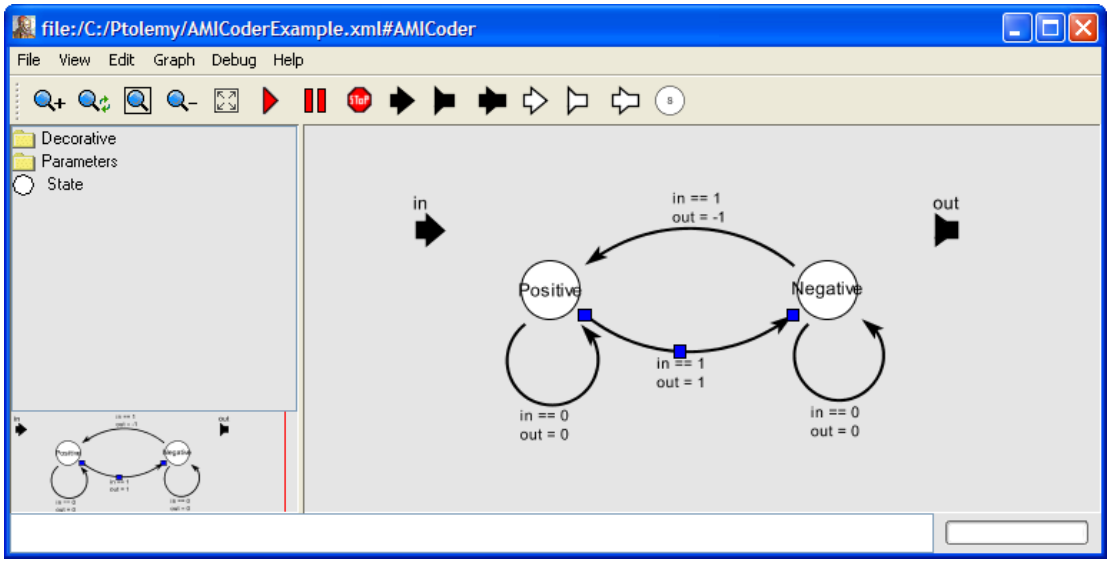


FIGURE 4.2. Vergil FSM editor showing the AMICoder.

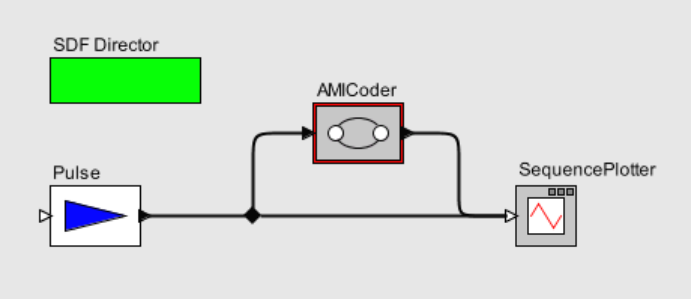


FIGURE 4.3. An SDF model with the AMICoder.

17. Edit parameters of the Pulse actor: set indexes to `{0, 1, 2, 3, 4, 5}`; set values to `{0, 1, 1, 1, 0, 1}`.
18. The model construction is complete.
19. Select View -> Run Window from the menu. Set director iterations to `6` and execute the model. For a better display of the result, open the set plot format dialog box, unselect connect and use various marks.

4.3 The Implementation of FSMActor

The FSMActor-related classes in the FSM kernel package are shown in figure 4.4.

The FSMActor class extends the CompositeEntity class and implements the TypedActor interface. An FSM actor contains states and transitions. The State class is a subclass of ComponentEntity. A State has two ports: incomingPort, which links to incoming transitions to the state, and outgoingPort, which links to transitions going out from the state. The Transition class is a subclass of ComponentRelation. A transition links to exactly two ports: the outgoing port of its source state, and the incoming port of its destination state.

4.3.1 Guard Expressions

The guard of a transition is specified by its *guardExpression* string attribute. Guard expressions are parsed and evaluated using the Ptolemy II expression language (see the Expressions chapter and the Data chapter for details). Guard expressions should evaluate to a boolean value. A transition is enabled if its guard expression evaluates to true. Parameters of the FSM actor and input variables (defined below) can be used in guard expressions.

Input variables represent the status and input value for each input port of the FSM actor. If the input port is a single port, two variables are used: a status variable named *portName*`_isPresent`, and a value variable named *portName*. If the input port is a multiport of width *n*, *2n* variables are used, two for each channel: a status variable named *portName_channelIndex*`_isPresent`, and a value variable named *portName_channelIndex*. The status variables will have boolean value true if there is a token at the corresponding input, or false otherwise. The value variables have the same type as the corresponding input, and contain the token received from the input, or null if there is no token. All input variables are contained by the FSM actor.

In the following examples (and the examples in the next section), we assume that the FSM actor has two input ports: a single port *in1* and a multiport *in2* of width 2; an output port *out* that is a multi-

port of width 2; and a parameter *param*.

- Guard expression: `in2_0 + in2_1 > 10`. If the inputs from the two channels of port *in2* have a total greater than 10, the transition is enabled. Note that if one or both channels of port *in2* do not have a token when this expression is evaluated, an exception will be thrown.
- Guard expression: `in1_isPresent && in1 > param`. If there is input from port *in1* and the value of the input is greater than *param*, the transition is enabled.

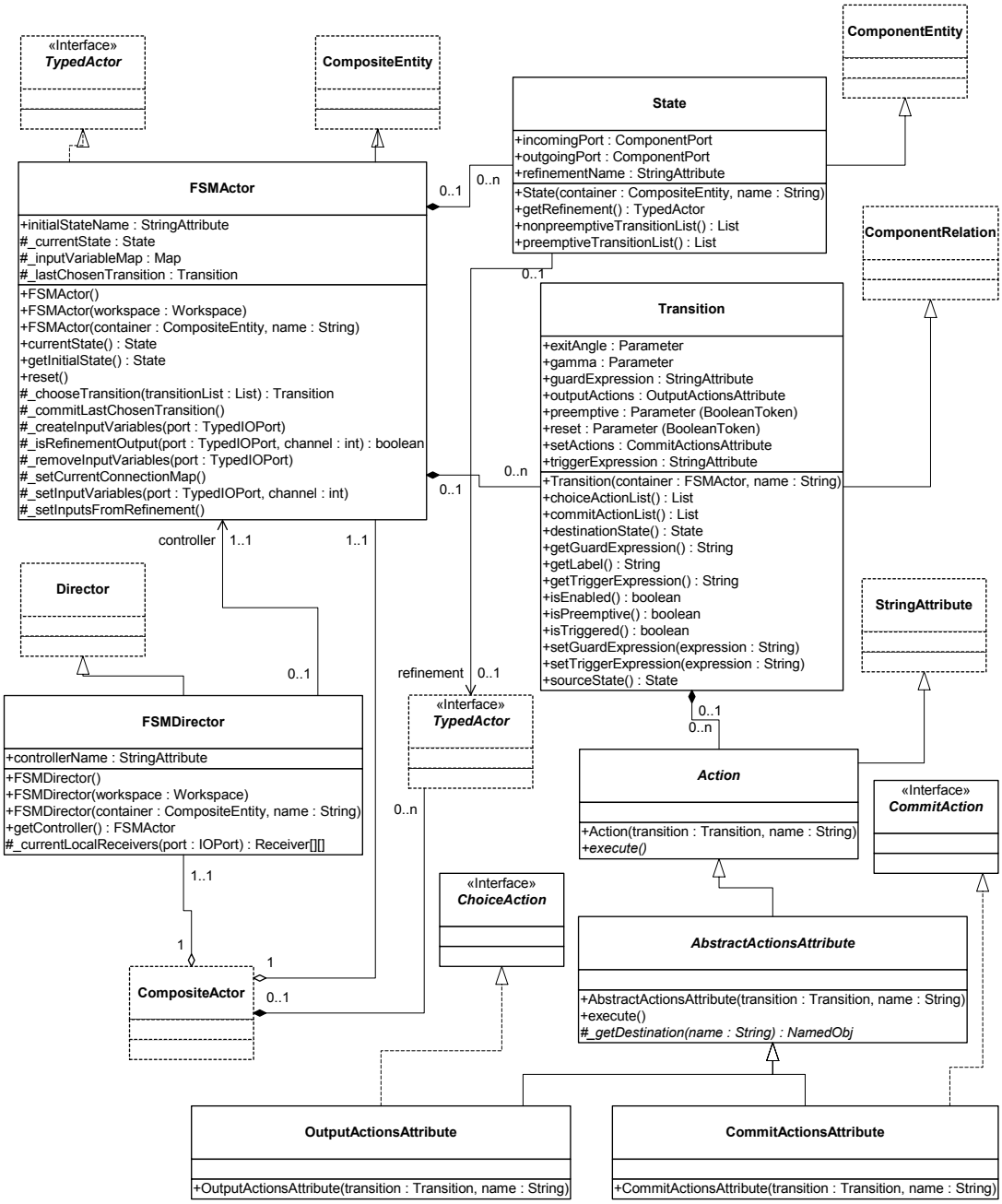


FIGURE 4.4. The UML static structure diagram of FSMActor-related classes.

4.3.2 Actions

A transition can have a set of actions that produce output tokens or set parameters of the FSM actor. To make FSM actors domain polymorphic (see section 4.5), especially for them to be operational in domains having fixed-point semantics, two kinds of actions are defined: choice actions and commit actions. Choice actions do not modify the extended state[1] of the FSM actor. They are executed when the FSM actor is fired and the containing transition is enabled. Commit actions may modify the extended state of the FSM actor. They are executed in postfire() if the containing transition was enabled in the last firing of the FSM actor. Two marker interfaces are defined in the FSM kernel package: ChoiceAction, which is implemented by all choice action classes, and CommitAction, implemented by all commit action classes.

A transition has an *outputActions* attribute which is an instance of OutputActionsAttribute. The OutputActionsAttribute class allows the user to specify a list of semicolon separated output actions of the form `destination = expression`. The expression can use parameters of the FSM actor and input variables. The destination is either a port name, in which case the result token from evaluating the expression is broadcast to all channels of the port, or of the form `portName(channelIndex)`, in which case the result token is sent to the specified channel. Output actions are choice actions.

- outputActions: `out = in1_isPresent ? in1 : 0`. Broadcast the input from port *in1*, or 0 if there is no input from *in1*, to the two channels of *out*.
- outputActions: `out(0) = param; out(1) = param + 1`. Send the value of *param* to the first channel of *out*, and the value of *param* plus 1 to the second channel.

A transition has a *setActions* attribute which is an instance of CommitActionsAttribute. The CommitActionsAttribute class allows the user to specify a list of semicolon separated commit actions of the form `destination = expression`. The expression can use parameters of the FSM actor and input variables. The destination is a parameter name.

- setActions: `param = param + (in1_isPresent ? in1 : 0)`. The input values from port *in1* are accumulated in *param*.

It is worth noting that parameter values are persistent. If not properly initialized, the parameter *t* in the above example will retain its accumulated value from previous model executions. A useful approach is to build the FSM model such that the initial state has an outgoing transition with guard expression `true`, and use the set actions of this transition for parameter initialization.

4.3.3 Execution

The methods that define the execution of an FSM actor are implemented as follows:

- `preinitialize()`: create receivers and input variables for each input port; set current state to the initial state as specified by the *initialStateName* attribute.
- `initialize()`: perform domain-specific initialization by calling the initialize(Actor) method of the director. Note that in the example given in section 4.2.1, the director will be the SDF director.
- `prefire()`: always return true. An FSM actor is always ready to fire.
- `fire()`: set the values of input variables; choose the enabled transition among the outgoing transitions of the current state; execute the choice actions of the chosen transition.
- `postfire()`: execute the commit actions of the last chosen transition; change state to the destina-

1. The extended state of an FSM actor is the current state of the state machine it contains plus the set of current values of its parameters.

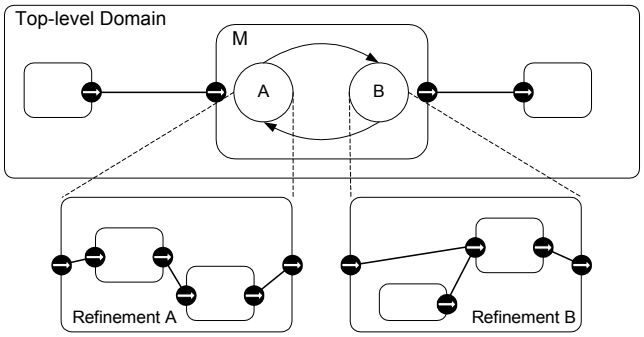


FIGURE 4.5. A modal model example.

tion state of that transition. If the destination state is a final state as specified by the *finalStateNames* attribute, then return false to inform the director not to fire this actor again.

Non-deterministic FSMs are not allowed[1]. The fire() method checks whether there is more than one enabled transition from the current state. An exception is thrown if there is. In the case when there is no enabled transition, the FSM will stay in its current state.

4.4 Modal Models

The FSM domain supports the *charts formalism with modal models. The concept of modal model is illustrated in figure 4.5. *M* is a modal model with two operation modes. The modes are represented by states of an FSM that controls mode switching. Each mode has a refinement that specifies the behavior of the mode. In Ptolemy II, a modal model is constructed in a modal model actor having the FSM director as local director. The modal model actor contains a mode controller (an FSM actor) and a set of actors that model the refinements. The FSM director mediates the interaction with the outside domain, and coordinates the execution of the refinements with the mode controller.

4.4.1 A Schmidt Trigger Example

In this section, we will illustrate how to build a modal model in Ptolemy II with a simple Schmidt trigger example. The output from the Schmidt trigger will move from -1.0 to 1.0 when its input becomes greater than 0.3, and will move back to -1.0 once its input becomes less than -0.3.

1. Open a Vergil graph editor. From Actors/HigherOrderActors, drag a ModalModel actor to the graph, rename it SchmidtTrigger. Add an input port named *in* and an output port named *out* to it.
2. Look inside SchmidtTrigger. This will open an FSM editor for the mode controller. Construct a two-state FSM as shown in figure 4.6. Set the *reset* parameter of both transitions to `true`. Set ini-

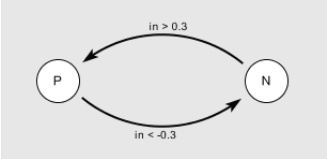


FIGURE 4.6. The mode controller for SchmidtTrigger.

1. This may change in future developments.

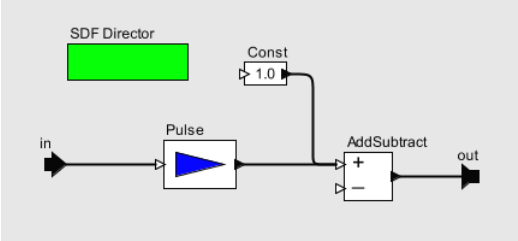


FIGURE 4.7. Model for the refinements in SchmidtTrigger.

tial state name of the mode controller to `N`.

3. Right click on the state named `P`, select Add Refinement. Specify the name of the refinement as RefinementP. A Vergil graph editor will be opened for the refinement. Build a model for it as shown in figure 4.7. Set the value of Const to `1.0`. Edit parameters of Pulse: set indexes to `{0, 1, 2, 3, 4}`, and values to `{-2.0, -1.6, -1.2, -0.8, -0.4}`.

4. Add a refinement named RefinementN to state `N`. Build a model for it similar to the one shown in figure 4.7. Set the value of Const to `-1.0`. Edit parameters of Pulse: set indexes to `{0, 1, 2, 3, 4}`, and values to `{2.0, 1.6, 1.2, 0.8, 0.4}`.

5. Back to the graph editor opened in step 1. Build the model as shown in figure 4.8. The model generates an input signal (a sinusoid plus Gaussian noise) for the SchmidtTrigger and plots its output. Edit parameters of Ramp: set init to `-PI/2`, and step to `PI/20`. Edit parameters of Gaussian: set standardDeviation to `0.2`.

6. Run the model for 200 iterations. A sample result is shown in figure 4.9.

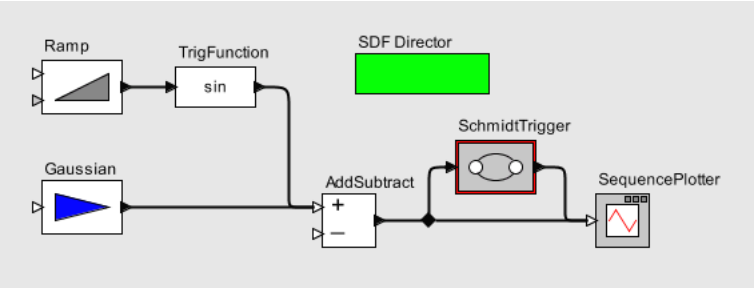


FIGURE 4.8. The top-level model with the SchmidtTrigger.

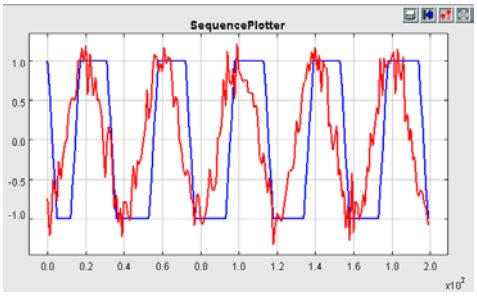


FIGURE 4.9. Sample result of the model shown in figure 4.8.

4.4.2 Implementation

The classes in the FSM kernel package that support modal models are shown in figure 4.10. The execution of a modal model is summarized below.

When a modal model is fired:

1. The FSM director transfers the input tokens from the outside domain to the mode controller and to the refinement of its current state.
2. The preemptive transitions from the current state of the mode controller are examined. If there is an enabled transition, execute the choice actions of the transition, go to step 5.
3. Fire the refinement of the current state.
4. The non-preemptive transitions from the current state of the mode controller are examined. If there is an enabled transition, execute the choice actions of the transition.
5. Any output token produced by the mode controller or the refinement is transferred to the outside domain.

To make a transition preemptive, set its *preemptive* parameter to true. The mode controller does not change state during successive firings in one iteration in order to support outside domains that iterate to a fixed point. In postfire(), if there is an enabled transition in the latest firing:

1. Execute the commit actions of the transition.
2. Set the current state of the mode controller to the destination state of the transition.
3. If the value of the *reset* parameter of the transition is true, the refinement of the destination state is initialized.

4.4.3 Applications

Hybrid System Modeling. An HSDirector class that extends the FSMDirector class is created for modeling hybrid systems with FSMs and continuous-time (CT) models. An example is presented in section

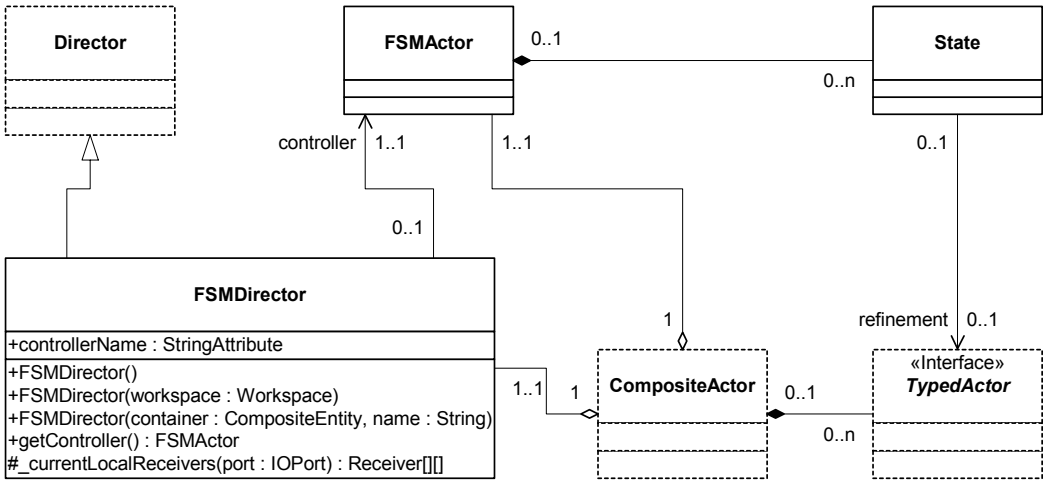


FIGURE 4.10. FSM kernel classes that support modal models.

2.7.3. Execution control is discussed in section 2.8.6.

Communication Protocol Modeling. Hierarchical FSMs are used to model protocol control logic. The timing characteristics of the communication channel are captured by discrete-event (DE) models. We have applied this approach to the alternating bit protocol. The detailed models can be found in the FSM domain demo directory ($PTII/ptolemy/domains/fsm/demo/ABP).