Interaction of Finite State Machines and Concurrency Models

Bilung Lee and Edward A. Lee
{bilung, eal}@eecs.berkeley.edu
University of California at Berkeley

Abstract

Hierarchical concurrent finite state machines (HCFSMs) dramatically increase the usability of finite state machines (FSMs). However, most formalisms that support HCFSMs, such as Statecharts (and its variants), tightly integrate the concurrency semantics with the FSM semantics. We, in contrast, allow FSMs to be hierarchically combined with multiple concurrency models, enabling selection of the most appropriate concurrency semantics for the problem at hand. A key issue for the success of this scheme is to define how FSMs interact with various concurrency models without ambiguities. In this paper, we focus on the interaction of FSMs and three concurrency models: synchronous dataflow, discrete-event and synchronous/reactive models.

1. Introduction

Hierarchical concurrent finite state machines (HCFSMs) increase their usefulness for the development of control-oriented systems by extending finite state machines (FSMs) with structuring and communication mechanisms. Hierarchy allows a state of the FSM to be refined into another FSM, i.e. a set of substates. Concurrency allows a state to be further decomposed into multiple simultaneously active FSMs that communicate through messaging of some sort.

A popular and seminal representative of the HCFSM model was introduced as the Statecharts formalism [7]. Since then, a number of variants [12] have been explored and exhibit different concurrency semantics. However, they tightly integrate the concurrency semantics with the FSM semantics. For example, the FSM is integrated with a synchronous/reactive concurrency model [1] in the Argos language [11], or a discrete-event concurrency model [3] in the co-design finite state machines (CFSM) model [4].

In fact, we observe that FSM, concurrency and hierarchy can be orthogonal semantic properties:

- FSM: This specifies the sequential behavior of a system in the form of states and transitions.
- Concurrency: This specifies the interaction between multiple simultaneous components and modules.
- Hierarchy: This specifies the interaction between a module and the refining components in that module.

In this paper, we advocate decoupling the concurrency semantics from the FSM semantics. By equipping the basic FSM with hierarchy and heterogeneity, hierarchical combinations of FSMs with various concurrency models become feasible. Key to this approach is to clearly define the interaction between different models before they can be combined together.

We begin by adding hierarchy and heterogeneity to the basic FSM, and explaining how FSMs are combined with concurrency models. Then, in particular, we discuss the interaction between FSMs and synchronous dataflow, discrete-event and synchronous/reactive concurrency models. Finally, we use a directed loop of two FSMs as an example to illustrate different concepts of communication in the three concurrency models.

2. Finite state machines

An FSM consists of a set of input events, a set of output events, a set of states, an initial state and a set of transitions. Consider an example of the FSM, shown in figure 1, with input events $\{a, b\}$ and output events $\{u, v\}$, where an event is a named variable that is either *present* or *absent*. Each elliptic node represents a state and each arc represents a transition. The arc without a source state points to the initial state, i.e. state α. Each transition links a source

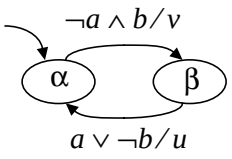

Figure 1. A basic FSM.

state with a destination state, and is labeled by either "*guard*/*action*" or "*guard*" (i.e. the action is omitted). A guard is a boolean expression over the input events. The evaluation of an event is either **true** or **false** when the event is either present or absent. The operators $\neg$, $\vee$ and $\wedge$ in guards correspond to the boolean operators **not**, **or** and **and**, respectively. An action lists a subset of the output events.

In one *reaction* of the FSM, a subset of the input events are present. One transition is triggered when its guard is **true** under the current input events. The FSM goes to the destination state of the triggered transition, and emits each output event in the action of the triggered transition, making these output events present. If the action is omitted, it means that no output event is emitted. An action only lists the output events to be emitted, and thus all other output events are absent.

2.1. Hierarchy

In a hierarchical FSM, a state may be refined into another FSM. With respect to the inner FSM called the *slave*, the outer FSM is called the *master*. Moreover, if a state is refined, it is called a *hierarchical* state; otherwise, the state is called an *atomic* state. For example, we can let the state β in figure 1 be refined into another FSM but let the state α not be refined, as illustrated in figure 2.

The hierarchy semantics define how the slave reacts relative to the reaction of its master. A reasonable semantics defines one reaction of the hierarchical FSM as follows: if the current state is an atomic state, the hierarchical FSM behaves just like a basic FSM. If the current state is a hierarchical state, then first the corresponding slave reacts, and then the master reacts.

2.2. Heterogeneity

Our hierarchical FSM is easily extended to support heterogeneity. The slave of a hierarchical state need not be an FSM. The key principle is that the slave must have a well-defined terminating computation that reacts to input events by (possibly) asserting output events. Therefore, the slave could be, for example, a Turing machine (that halts), a C procedure (that eventually returns), a dataflow graph (with a well-defined iteration), etc. It can even be concurrent. In this paper, we focus on combinations of FSMs with concurrency models.

The hierarchy semantics is similarly defined as in the previous section with one subtle modification: If the current state is a hierarchical state, then first the corresponding slave is invoked and then the master reacts. When the slave is invoked, it performs a determinate and finite operation, called a *step* of the slave, which reacts to input events and may assert output events. One step of a slave FSM is one reaction of the FSM.

2.3. Hierarchical combination

With support of hierarchy and heterogeneity, the FSM can be combined with almost any concurrency model. Our objective is the hierarchical nesting of the FSM with concurrency models, as shown in figure 3. We schematically illustrate the modules of the concurrency model with rectangular blocks and the states of the FSM model with elliptic nodes. The depth and order of the nesting is arbitrary.

To achieve the goal, first we need for an FSM to be able to describe a module in a concurrency model. For example, in figure 4, two FSMs are embedded inside the modules of a concurrency model. This can be done as long as that model provides a way to determine the input events and when a reaction should occur for each FSM. Most interestingly, the two FSMs are concurrent FSMs based on the concurrency semantics provided by that model.

On the other hand, a state of an FSM needs to be able to be refined into a concurrency subsystem, as explained above in section 2.2.

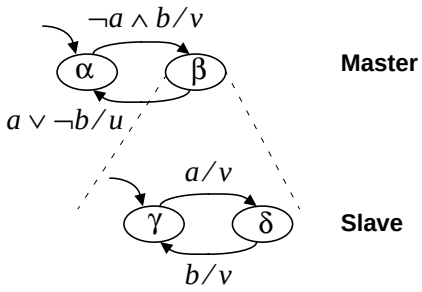

Figure 2. A hierarchical FSM.

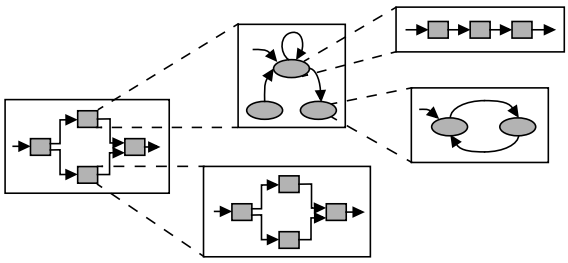

Figure 3. Hierarchical nesting of FSMs with concurrency models.

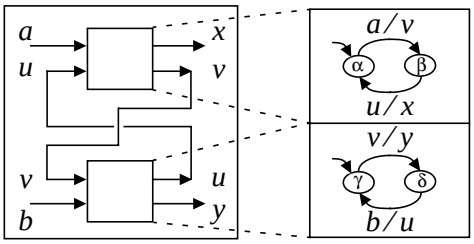

Figure 4. Two FSMs are embedded inside the modules of a concurrency model.

3. Interaction with concurrency models

Different models capture different semantic properties of a system. When different models are combined together, differences between them can lead to ambiguities. These need to be resolved by defining the semantics of interaction between different models. Models that support concurrency are numerous. In this paper, we focus on the interaction semantics of FSMs with synchronous dataflow (SDF), discrete-event (DE) and synchronous/reactive (SR) concurrency models.

3.1. Synchronous dataflow

Under the SDF paradigm [10], a system consists of a set of blocks interconnected by directed arcs. The blocks represent computational functions that map input data into output data when they *fire*. The arcs represent streams of data *tokens*, and can be implemented as first-in-first-out queues. Upon firing, a block consumes a fixed number of tokens from each input arc and produces a fixed number of tokens on each output arc. The number of tokens consumed and produced can be used to unambiguously define an *iteration*, or minimal set of firings that return the queues to their original size. Thus, the firing schedule for an iteration can be determined at compile time.

3.1.1. FSM inside SDF

When an FSM describes a block of an SDF graph, it must externally obey SDF semantics. Hence, it must consume and produce a fixed number of tokens on every input and every output. We relate one firing of the SDF block to one reaction of the embedded FSM.

A slight subtlety for the SDF is that absence of a token is not a well-defined, testable condition. Thus, the absence of an event in FSM must appear explicitly as a token in SDF. A simple approach is to encode presence and absence using boolean-valued tokens. I.e. a true-valued token means the event is present and a false-valued token means it is absent.

3.1.2. SDF inside FSM

When an SDF graph refines a state of an FSM, one step of the slave SDF graph is taken to be one iteration. In other words, when the refined state is the current state, the hierarchical FSM reacts with one iteration of the slave SDF graph followed by one reaction of the master FSM.

If the slave SDF graph consumes or produces more than one token on inputs or outputs, the semantics becomes more subtle. Suppose for example that a system consists of three-level hierarchy: A slave SDF graph refines a state of an FSM that describes a block of another SDF graph. In this case, the FSM needs to inform its outer SDF graph how many tokens are consumed or produced by its inner SDF graph because the outer SDF graph requires this information to compute the firing schedule. Furthermore, there may be more than one state of the FSM refined into the SDF graph. Thus, all slave SDF graphs are required to consume or produce the same number of tokens on each input or output.

3.2. Discrete events

The DE model [3] carries a notion of *global time* that is known simultaneously throughout the system. An event occurs at a point in time. In a simulation of such a system, each event needs to carry a *time stamp* that indicates the time at which the event occurs. The time stamp of an event is typically generated by the block that produces the event, and is determined by the time stamp of input events and the latency of the block. The DE simulator needs to maintain a global event queue that sorts the events by their time stamps. It defines the *current time* of the system to be the smallest time stamp in the event queue, and chronologically processes each event by sending it to the appropriate block, which reacts to the event (*fires*).

3.2.1. FSM inside DE

An FSM embedded inside a DE block performs one reaction when the DE block fires, which occurs when there is an event present at one of its inputs. Unlike SDF, the notion of presence and absence of an event is the same in DE and FSM. However, in DE, every event needs a time stamp, something not provided by the FSM. We choose the semantics where the FSM appears to the DE as a *zero-delay* block. I.e. the event passed to a DE system in a reaction of the FSM is assigned the same time stamp as the input event that triggers that reaction. Nevertheless, we

may explicitly connect the FSM-embedding block with a delay block to simulate the delay occurring inside the FSM subsystem.

3.2.2. DE inside FSM

When a DE model refines a state of an FSM, one step of the slave DE subsystem is the simulation of that subsystem until its current time matches the time stamp of its input events. In particular, if previous FSM describes a block of another DE model, the events passed to invoke the inner DE subsystem by the FSM will have the current time of the outer DE system as their time stamps. Thus, the notion of current time keeps consistent throughout all DE models in the hierarchy. Moreover, the semantics need not impose other consistency constraint, as we had to do with SDF, even when more than one state of the FSM is refined into a DE model.

3.3. Synchronous/Reactive models

An SR system [1] is a set of blocks instantaneously communicating through unbuffered directed arcs. Execution of the system occurs at a sequence of discrete *instants*. To ensure that the system is deterministic, i.e. always finding the same behavior given the same inputs, a partial order relation is imposed on the arc values that is augmented with a bottom value $\perp$ interpreted as "unknown", and the functions computed by the blocks are required to be monotonic with respect to this partial order relation. Together, these allow the system behavior at each instant to be defined as the least fixed point of the composition of all block functions.

Most familiar functions are strict functions that are always monotonic. However, a directed loop of all strict functions always causes causality problem - the least fixed point is all unknown. The use of non-strict functions allows directed loops with less trivial solutions.

3.3.1. FSM inside SR

To make best use of directed loops in an SR system, the FSMs need to be treated as non-strict functions in each reaction. For example, suppose that there are two outgoing transitions, labeled as "$a \wedge b / x$" and "$a \wedge \neg b / x$", for a state of an FSM. We can see that the function mapping the inputs a and b into the output x at that state is simply $f_x(a, b) = (a \wedge b) \vee (a \wedge \neg b) = a$. In other words, as long as input a is known to be present or absent, the output x can be asserted without knowing input b. This analysis can be automated to get a simplified function for each output at each state. Then, these simplified functions indicate for each state what inputs need to be known to define an output.

3.3.2. SR inside FSM

Embedding SR systems inside the states of the FSMs is straightforward. When a state of the FSM is refined into an SR subsystem, the semantics of SR are simply exported to the outer model in which the FSM is embedded. Moreover, one step of the slave SR subsystem is taken to be one instant.

4. Comparison of communication concepts

In the combination, concurrent FSMs are achieved by embedding FSMs inside a concurrency model. Most interestingly, they exhibit different communication mechanisms in different concurrency models. In particular, we use a directed loop of two FSMs in a concurrency model, depicted in figure 5, as a comparison.

First, if the concurrency model is SDF, then the FSM **A** and the FSM **B** are both waiting for one token from each other. This is called *deadlock* [10]. However, this deadlock can be avoided by adding at least one unit delay (an initial token) on either arc in the directed loop, allowing one FSM to fire first. Suppose that a unit delay exists on the arc from the FSM **B** to the FSM **A**. Reacting to a present event (a true-valued token) on input *a*, the FSM **A** makes transition from state α to state β and emits the event *v*, and then the FSM **B** makes transition back to state γ and emits the event *u*. However, due to the unit delay, the event *u* will not be fed back to the FSM **A** until the next iteration. In other words, the communication exhibits *delayed semantics* on the delay arc in an SDF graph.

Unlike in SDF, directed loops with zero-delay are always permitted in DE and intrinsically exist in SR. However, they have different interpretations and may exhibit different behaviors in the two models. When the concurrency model in figure 5 is DE, it will start a sequence of firings (FSM **A**, FSM **B** and then FSM **A**) reacting to a

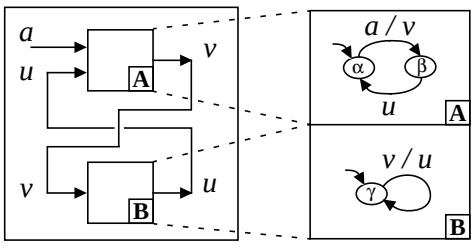

Figure 5. A directed loop of two FSMs in a concurrency model.

present event on input *a*. The communication between the two FSMs can be interpreted as a sequence of *micro steps* within a *macro step* delimited by the occurrence of the external input event *a*. In other words, the transitions of the FSMs strictly follow each other as micro steps within a macro step. This interpretation resembles the *micro-step semantics* in [8].

Now consider the concurrency model in figure 5 to be an SR system. When the input event on *a* is present at a certain instant, after the least fixed point is found (both events *u* and *v* are present), the two FSMs make transition according to the least fixed point. Under this *fixed-point semantics*, all generated events and triggered transitions are considered genuinely simultaneous within the same instant. Note that, unlike in the micro-step semantics, each FSM in the directed loop can only make a transition once reacting to an external input event *a*.

5. Conclusions

We have explored the combination of FSMs with three concurrency models, namely synchronous dataflow, discrete-event and synchronous/reactive models. The advantages for this system specification scheme are

- Heterogeneous: Diverse models can coexist and interact by hierarchical combination.
- Modular: Distinct portions of a system can be separately modeled, choosing the best appropriate modeling technique.
- Extensible: Additional concurrency models can be included in the combination as long as we provide the interaction between different models.

The above approach is under implementation in Ptolemy [2]. We have implemented an FSM domain [9], and have integrated it with two existing concurrency domains, the SDF and the DE domains. The next stage of implementation is to integrate the SR domain [5] with the FSM domain.

The FSMs discussed are "pure", in the sense that events cannot carry values other than presence or absence. In many applications, non-boolean values are more useful. We refer reader to [6] for details about supporting valued FSMs.

Finally, in our scheme, the semantics of FSM, concurrency and hierarchy are naturally supported in a manner similar to HCFSMs. However, in fact, the typical applications of HCFSMs do not really illustrate the main advantages of our approach. A signal processing system would be a better illustration, where FSM subsystems are used for control logic and dataflow subsystems are used for numeric-intensive signal processing.

Acknowledgments

This research is part of the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the State of California MICRO program, and the following companies: Cadence Design Systems, Hewlett Packard, Hitachi, Hughes Space and Communications, NEC, and Philips.

References

[1] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1270-1282, September 1991.

[2] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *Int. Journal of Computer Simulation*, special issue on "Simulation Software Development," vol. 4, pp. 155-182, April 1994.

[3] C. Cassandras, "Discrete Event Systems, Modeling and Performance Analysis," Irwin, Homewood, IL, 1993.

[4] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli, "Hardware-Software Codesign of Embedded Systems," *IEEE Micro*, pp. 26-36, August 1994.

[5] S. A. Edwards, "The Specification and Execution of Heterogeneous Synchronous Reactive Systems," Ph.D. dissertation, UCB/ERL M97/31, Electronics Research Laboratory, University of California, Berkeley, May 1997.

[6] A. Girault, B. Lee, and E. A. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models," April 13, 1998 (revised from UCB/ERL M97/57, Electronics Research Laboratory, University of California, Berkeley, August 1997).

[7] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Sci. Comput. Program.*, vol 8, pp. 231-274, 1987.

[8] D. Harel, A. Pnueli, J. Schmidt, and R. Sherman, "On the Formal Semantics of Statecharts," *Proc. of the Symposium on Logic in Computer Science*, p. 54-64, June 1987.

[9] B. Lee and E. A. Lee, "Hierarchical Concurrent Finite State Machines in Ptolemy," In *Int. Conf. on Application of Concurrency to System Design*, Fukushima, Japan, March 1998.

[10] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Trans. on Computers*, January 1987.

[11] F. Maraninchi, "Operational and compositional semantics of synchronous automaton compositions," *Proc. of 3rd Int. Conf. on Concurrency Theory*, pp. 550-564, August 1992.

[12] M. von der Beeck, "A Comparison of Statecharts Variants," *Proc. of Formal Techniques in Real Time and Fault Tolerant Systems*, pp. 128-148, September 1994.