Fusing Dataflow with Finite State Machines

Bilung Lee

MS Report

Department of Electrical Engineering and Computer Science

University of California

Berkeley, California 94720

ABSTRACT

The dataflow model of computation has been used extensively in signal processing design, and is particularly convenient for numeric-intensive computation of applications. Finite state machines (FSMs) have been developed to solve a different class of problems, namely sequential control. In this project, we propose to hierarchically nest the dataflow and FSM models of computation. With the two models mixed, concurrency and hierarchy are naturally supported in a manner similar to hierarchical FSMs, like Statecharts. This provides a clean and simple mechanism for describing systems that combine sophisticated signal processing with sophisticated control. We implement the ideas in the Ptolemy software environment, which has been under development at University of California at Berkeley.

Acknowledgment

The work that leads to this paper would not be possible without the assistance from my advisor, Edward A. Lee. I would also like to thank the whole Ptolemy team for building such a magnificent environment for experimenting the concepts discussed in this paper. In particular, I wish to thank Wan-teh Chang.

Table of Contents

1.0 Introduction 5

2.0 Models of Computation 6

2.1 Dataflow 6

2.2 Finite state machines (FSMs) 6

2.2.1 Basic FSMs 6

2.2.2 Hierarchical FSMs (HFSMs) 7

2.3 Mixing Dataflow with FSMs 9

3.0 Ptolemy Implementation 11

3.1 Overview of Ptolemy 11

3.2 Wormholes 12

3.3 Dataflow Domains in Ptolemy 13

4.0 FSM Domain 15

4.1 Kernel 15

4.1.1 State Star 15

4.1.2 Target and Scheduler 16

4.2 Stars 17

4.2.1 Moore-type state (FSMMoore) 17

4.2.2 Mealy-type state (FSMMealy) 17

4.2.3 Data Input (FSMDataIn) 17

4.2.4 Data Output (FSMDataOut) 18

4.3 Interaction Semantics 18

4.3.1 FSM inside Dataflow 18

4.3.2 Dataflow inside FSM 18

4.3.3 Dataflow inside FSM inside Dataflow 19
4.4 Graphical User Interface (GUI) 20
5.0 Application example 20
5.1 System Description 21
5.2 Simulation 22
5.3 Discussion 26
6.0 Conclusion 26
7.0 Future Work 27
8.0 References 27

1.0 Introduction

Real-time embedded systems frequently need both numerical computations and sophisticated control. A digital telephone answering machine, for example, may contain a signal processing part including speech compression and decompression. Moreover, to develop such a machine, a sizable portion of effort is required to design the control flow that manages the initialization of the connection and the interaction between the caller and the machine. Therefore, the design process for such a system needs two different design methodologies.

The dataflow model [7] is useful for specifying the numeric-intensive systems, like most signal processing systems. However, it is far less convenient for representing the control-dominated systems, like real-time process controllers. On the other hand, the FSM model is well known for its ability to easily describe a control-oriented system. Mixing dataflow with FSMs is a good solution for representing a system which requires both signal processing and control. Our main objective is to get the best of both worlds while preserving the integrity and simplicity of each model. In other words, instead of creating a new semantics combining both models, we would like to develop a mechanism that works the two models together, but allows each model to retain its purity of semantics. The advantage of this is that it does not compromise the ability to synthesize efficient hardware and software implementations.

Ptolemy [3] is a software environment that supports heterogeneous system specification, design and simulation. It allows diverse models of computation coexisting and interacting. In Ptolemy, different models of computation are hierarchically nested, with strict information hiding between layers of the hierarchy. Therefore, Ptolemy serves as a good development environment to mix these two different kinds of computational models, dataflow and FSMs.

We define a prototype implemented in Ptolemy where FSMs and dataflow are hierarchically nested. Each model isolates its semantic properties from the other because of the information hiding in Ptolemy. With the two models nested, it becomes convenient to specify those applications that consist of both dataflow and control flow.

2.0 Models of Computation

A key principle to support heterogeneous design methodologies is the notion of *models of computation*. A model of computation is the semantics that defines the interaction between modules and components. For example, dataflow and FSMs are two distinct models of computation.

2.1 Dataflow

Dataflow is a particular type of process network model [7]. In dataflow, a program is specified by a directed graph. The nodes of the graph represent computational functions (*actors*) that map input data into output data when they *fire*, and the arcs represent the exchanged data (*streams* of *tokens*) from one node to another. The processes in a dataflow graph are executed by repeated actor firings according to *firing rules*. Variants of this model are used in many visual programming environments intended for signal processing, such as COSSAP from the Synopsys, the DSP Station from the Mentor Graphics, Khoros from the University of New Mexico [9], Ptolemy from the University of California at Berkeley, and SPW from the Alta Group of Cadence.

2.2 Finite state machines (FSMs)

2.2.1 Basic FSMs

An FSM model consists of a set of *states*, a set of *transitions* between states, and a set of *actions* associated with these states or transitions. Each transition is a function that determines the next state from the current state and the input, and each action is a function that determines the output from the current state and/or the input.

There are two distinct types of FSMs, where output is associated with the state (a Moore machine) and with the transition (a Mealy machine). A directed graph, called a *state transition diagram*, can be used to describe an FSM. Figure 1 shows state transition diagrams of equivalent Moore and Mealy machines. Each elliptic node represents a state and each arc from node to node represents a transition. In the Moore machine, the number adjacent to each arc represents the input value that triggers the transition, and the number after the slash in each node represents the output value in that state. Similarly, in the Mealy machine, the number before the slash adjacent to

each arc represents the input value that triggers the transition, and the number after the slash represents the output value associated with that transition.

In general, as shown in Figure 1, a Moore machine may require more states than an equivalent Mealy machine. This is because in a Mealy machine there may be more than one arc pointing to a single state, each arc with a different output value; however, in a Moore machine, each different output value requires one state.

The FSM model of computation is suitable for modeling control-dominated systems. However, the basic FSM model, which is flat and sequential, has a major weakness; nontrivial systems have a very large number of states.

2.2.2 Hierarchical FSMs (HFSMs)

The hierarchical FSM (HFSM) model of computation adds support for hierarchy and concurrency into the basic FSM model. Hierarchy permits each state to be further decomposed into a set of substates, and thus the complexity of the state space is reduced. Concurrency permits a further reduction of complexity by allowing multiple FSMs to operate simultaneously and communicate through signals.

Figure 2 shows an example of a simple three-bit counter represented by means of an HFSM. In this figure, we can see that the state **Counting** is decomposed into three concurrent components,

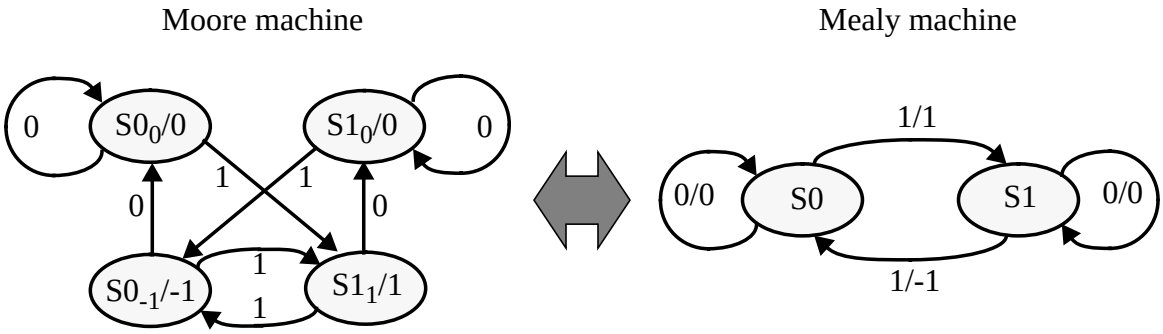


Figure 1. Equivalent Moore and Mealy machines.

A, **B** and **C**, each of them consisting of two states. Compared to the representation in a flat FSM model (see Figure 3), the HFSM model may reduce the complexity of the state space because of its hierarchy and concurrency.

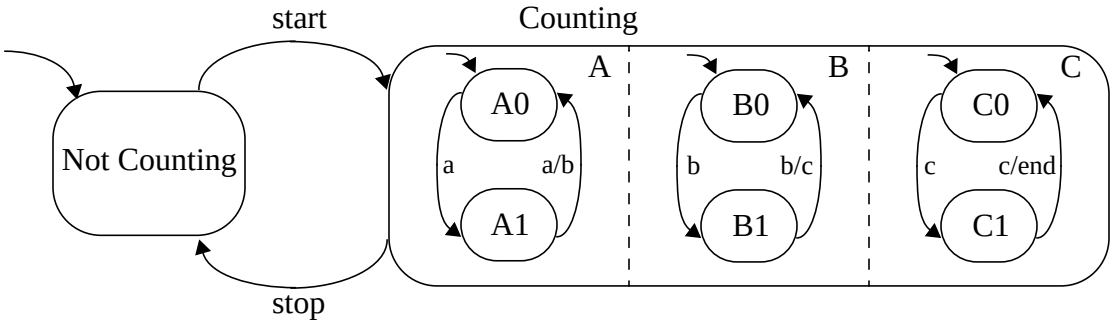


Figure 2. An HFSM representation of a 3-bit counter.

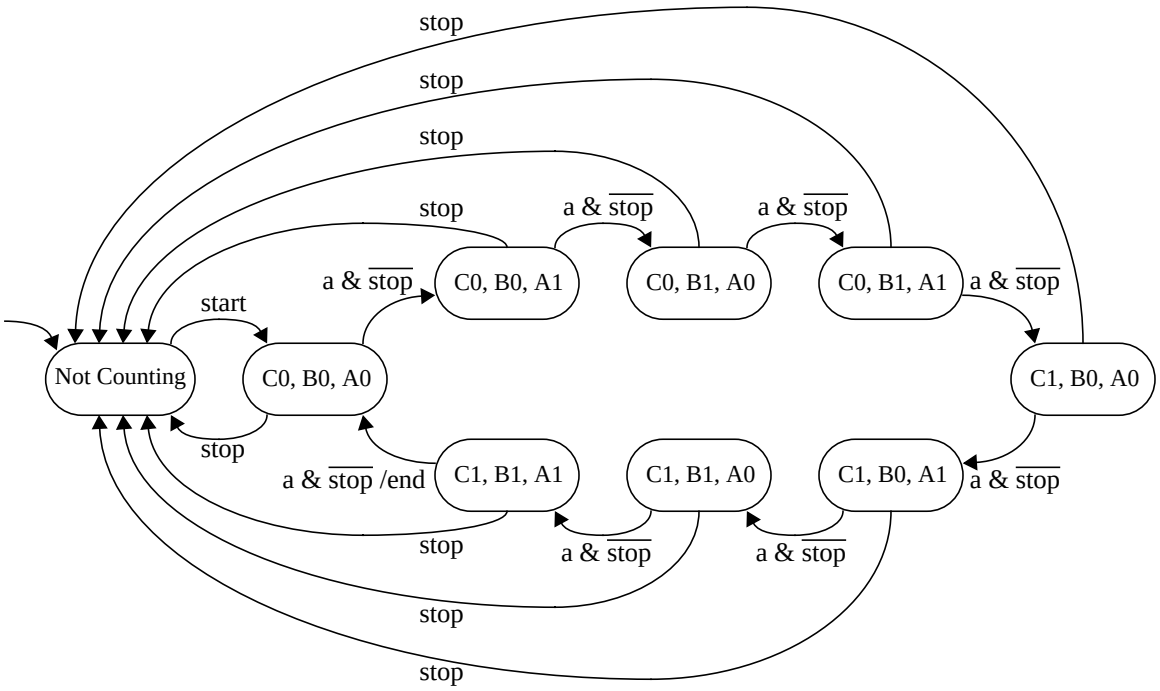


Figure 3. A basic FSM representation of a 3-bit counter.

The Statecharts formalism [4] and at least 20 variants [10], including Argos [6], are typical examples of HFSM models. Some programming environments use similar models, such as Statemate from iLogix [5], SpeedChart from Speed Electronics Inc., StateVision from Vista Technologies and VisualHDL from Summit Design Inc.

2.3 Mixing Dataflow with FSMs

We identify three orthogonal semantic properties in Statecharts and related models of computation: FSM, concurrency and hierarchy. After we suppress hierarchy-crossing transitions allowed in Statecharts (see Figure 4), two important observations are as follow: First, we get a simpler model in which the FSM semantics can be cleanly separated from the concurrency semantics. Second, the specification of concurrent FSMs in Statecharts can be considered as a syntactic shorthand for an interconnection of FSMs in a concurrency model (see Figure 5). In other words, the basic FSM model can be hierarchically mixed with various concurrency models to get many models that are similar to Statecharts. Although this lacks hierarchy-crossing transitions of Statecharts, those transitions are considered by many to violate modularity in hierarchical design anyway.

The concurrency semantics in parallel FSMs is one of the main differences between variants of Statecharts. Statecharts and most related formalisms use the notion of *instantaneous broadcast* to model the communication between concurrent FSMs. This means that all concurrent FSMs may contain transitions that are executed simultaneously in response to an input event, and these transitions may produce internal events that trigger other transitions instantaneously.

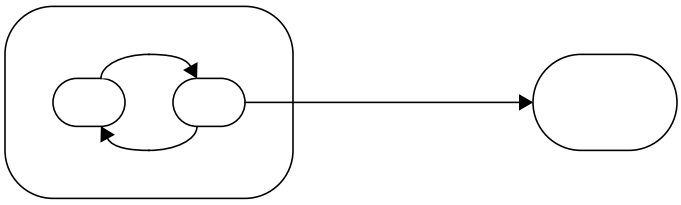


Figure 4. An example of a hierarchy-crossing transition allowed in Statecharts.

There are at least two interpretations of such “instantaneous broadcast”, *microsteps* and a *fixed point*. In the microsteps case, the transitions occurring at a given time instant have a natural order. In the fixed point case, they are genuinely simultaneous, and the execution of transitions involves finding a consistent value (called a *fixed point*) for all events at a given time instant. In the former case, the concurrency semantics can be specified using the dataflow model, and in the latter case, it can be specified using the synchronous/reactive communication model that is found in synchronous languages. In Figure 5, both interpretations lead the same result, so the concurrency property can be specified by a block diagram with the three interconnected blocks in a dataflow model or a synchronous/reactive communication model.

However, some concurrency models may not work in some situations. For example, a pair of transitions that produce events triggering each other (an *instantaneous dialog*) will cause a zero-delay loop in terms of block diagrams (see Figure 6), and *synchronous dataflow* (see Section 3.3), a typical type of dataflow model, does not allow zero-delay loops. On the other hand, the synchro-

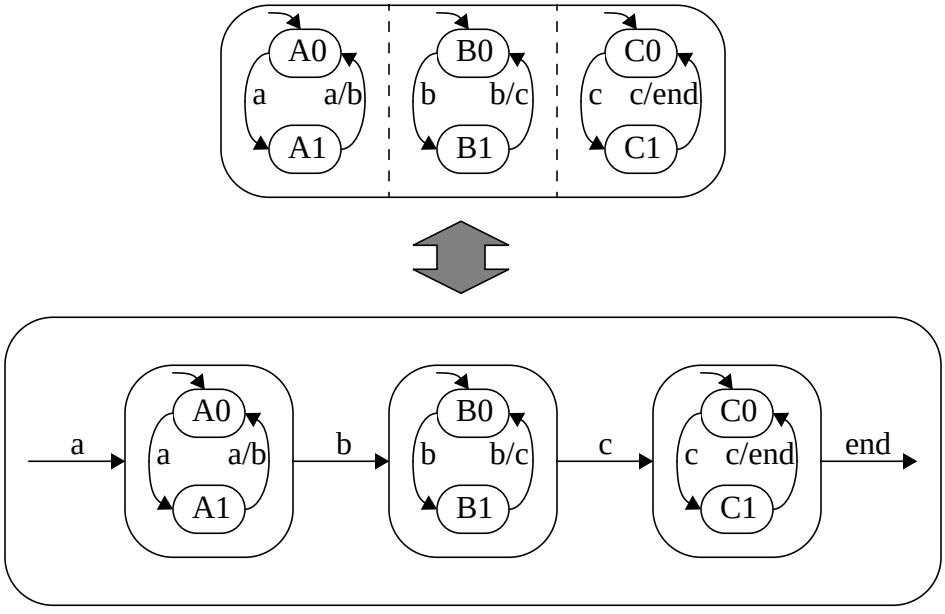


Figure 5. The specification of concurrent FSMs in Statecharts can be considered as a syntactic shorthand for an interconnection of FSMs in a concurrency model.

nous/reactive communication model allows the specification of zero-delay loops. We will focus on the dataflow model in this paper.

3.0 Ptolemy Implementation

3.1 Overview of Ptolemy

A system in Ptolemy is represented as a block diagram constructed by interconnecting both user-created and existing library blocks. Two types of blocks can be used for interconnection: the *Star* and the *Galaxy.* A Star is a fundamental block containing code segments for execution or code generation. A Galaxy is a block that internally contains Stars and possibly other Galaxies. By building a subsystem as a Galaxy, a large complicated system can be decomposed into many subsystems which are hierarchically nested and interconnected. A *Universe* is a complete Ptolemy application and describes a system.

The input and output interfaces in a block are called *Porthole*s. Interconnection of blocks is achieved by connecting the Portholes of blocks. Data objects passed between the blocks in Ptolemy are called *Particle*s.

A *Domain* encapsulates a type of model of computation in Ptolemy. Users can choose a *Target* for a Universe or a Galaxy in a specific Domain to define the mechanism by which a system is executed. Associated with a Target is a *Scheduler* which determines the operational order of each block in the application.

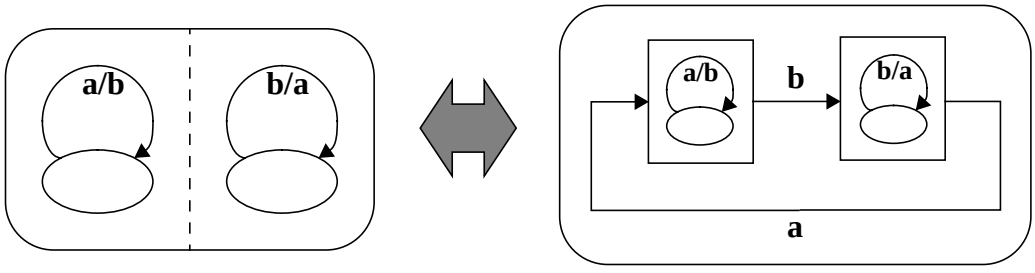


Figure 6. A pair of transitions that produce events triggering each other will cause a zero-delay loop in terms of block diagrams.

3.2 Wormholes

In Ptolemy, different domains are intermixed hierarchically to work together. In other words, two domains do not interact as peers. Instead, a domain may appear as a block inside another domain, as shown in Figure 7. Such a mechanism is a significant feature in Ptolemy and is called *Wormhole*. It encapsulates a subsystem specified in one domain within a system specified in another. The key idea of a Wormhole is that it must obey the semantics of outer domain at its boundary and the semantics of the inner domain internally.

We develop an FSM domain by generalizing the wormhole mechanism in Ptolemy. Each action, specifying the numerical computation and signal processing, of a module in the FSM domain can be defined by a subsystem in any domain, including FSM. Then, this subsystem is encapsulated as a Wormhole in the FSM domain. This means that a module in the FSM domain can be associated with any number of subsystems defined using different semantics in various domains. Moreover, this FSM module can be placed inside any domain such that various domains are hierarchically mixed. Figure 8 shows an example where the FSM and dataflow domains are nested.

Intuitively, when we put an FSM subsystem into another system in the FSM domain, the hierarchy property of an HFSM is easily achieved, as shown in Figure 9.

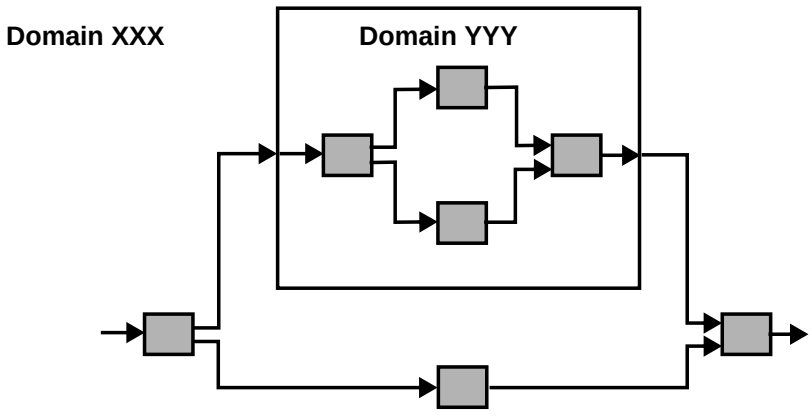


Figure 7. Mixing domains using hierarchy: a subsystem in domain YYY embedded in domain XXX as a block.

3.3 Dataflow Domains in Ptolemy

There are three existing dataflow domains in Ptolemy: *Synchronous dataflow* (SDF), *Boolean dataflow* (BDF) and *Dynamic dataflow* (DDF). We will focus on SDF mixed with FSM in this paper. Other domains can be explored in a similar way.

The SDF domain is one of the most mature domains in Ptolemy, and is an appropriate model for signal processing algorithms. In this domain, the firing order of the blocks is determined once, and may be repeated periodically during simulation runs. Each block consumes and produces a fixed number of data tokens on each input and output of the block at each firing.

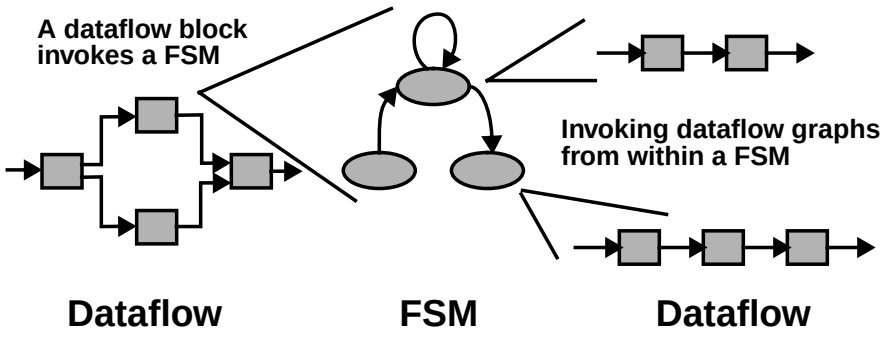


Figure 8. Hierarchical nesting of dataflow graphs with FSM controllers.

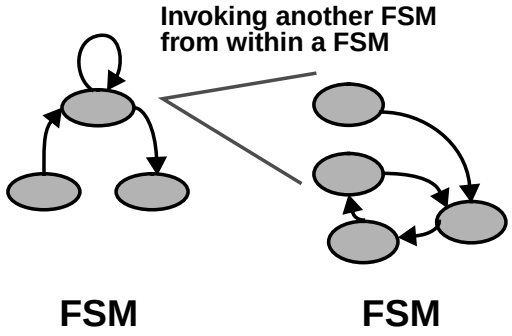
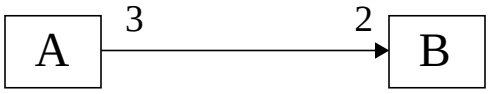


Figure 9. The hierarchy property of an HFSM can be achieved by nesting an FSM within another FSM.

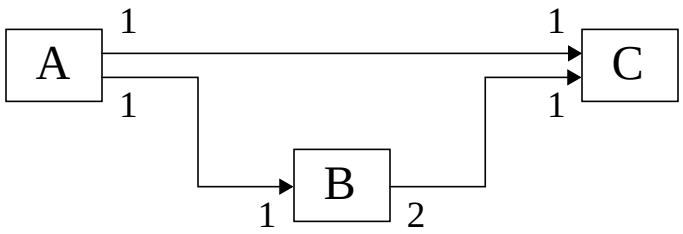
There is no notion of time in the SDF domain. The unit of work is given in the unit of SDF *iteration*. At each iteration, each block in the SDF graph fires the minimum number of times to satisfy the *balance equations* [7]. The balance equations for an SDF graph are the set of equations relating the number of tokens consumed to the number produced for each pair of blocks associated with an arc. Consider a simple SDF graph and its balance equation depicted in Figure 10. The number adjacent to the connection between an arc and a block represents the number of tokens consumed or produced in that block, and the unknowns r_A and r_B are the minimum firing times that are required to maintain balance on each arc for blocks A and B respectively. We can see that the solution for the balance equation is $r_A = 2$ and $r_B = 3$.

An SDF graph is said to be *inconsistent* and is flagged as an error if the balance equations have no non-zero solution, as shown in Figure 11. Another situation that an SDF graph is not valid is when there are zero-delay loops in the graph, as shown in Figure 12.



$$3r_A = 2r_B$$

Figure 10. A simple SDF graph and its balance equation.



$$r_A = r_C$$

$$r_A = r_B$$

$$2r_B = r_C$$

Figure 11. An inconsistent SDF graph and its balance equations.

4.0 FSM Domain

4.1 Kernel

4.1.1 State Star

There is a specific type of kernel class in the FSM domain: **FSMStateStar**. This is the base class for representing a state in the state transition diagram of the FSM. Important derived types of this class are **FSMMoore** (see Section 4.2.1), representing a state for the Moore-type FSM, and **FSMMealy** (see Section 4.2.2), representing a state for the Mealy-type FSM.

For a module in this domain, there may be many actions associated with it, and each action can be specified by a subsystem in any domain. Therefore, one of the jobs for this class is to create the required FSM wormhole to encapsulate the action subsystem.

Two key functions in this class are invoked from the scheduler. First, reacting to an input, the FSM needs to determine a transition to the next state from the possible transitions out of the current state. A transition is chosen if the condition associated with it is true. In Figure 13, for example, suppose that S0 is the current state, then when the input is equal to one, the condition associated with transition T1 becomes true, and the FSM makes a transition from S0 to S1. Moreover, we restrict our FSM to be *deterministic*; i.e. for each input, there exists *at most* one enabled transition out of each state. Furthermore, when no condition is true, we define it as a *reflexive* transition; i.e. the FSM makes a transition back to the current state.

Second, as mentioned earlier, there may be many actions associated with an FSM. Although an action is associated with a transition in a Mealy machine, in a sense, some actions can also be

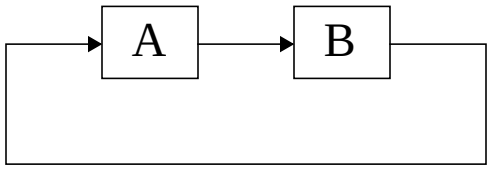


Figure 12. An SDF graph with a zero-delay loop.

viewed as attached to a state (see Section 4.2.2). Therefore, we can define a virtual function in this class, and thus a redefined function in the derived class is invoked to do the action depending on what type of machine it is.

4.1.2 Target and Scheduler

The machine type is specified as a parameter of the **FSMTarget**. This information will be passed to **FSMScheduler**. There are two main reactions to an input in **FSMScheduler**: changing state and performing an action. Whenever there is an input triggering the FSM, the scheduler will try to find the transition that matches the input and current state by invoking a function in the **FSMStateStar**. Once the next state is found, for a Moore-type machine, it makes the transition first, and then does the action in the state being entered; for a Mealy-type machine, it does the action associated with the transition, and then makes the transition.

Since the FSM domain has no notion of time, the unit of work reacting to an input is *one state transition* in an FSM. "*One state transition*" here means a transition from one state to another or back to the same state plus an action.

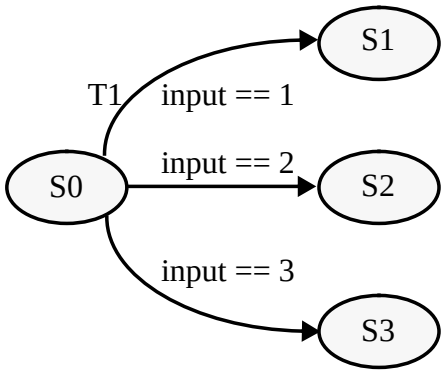


Figure 13. When the input is equal to one, this FSM makes a transition from S0 to S1.

4.2 Stars

4.2.1 Moore-type state (FSMMoore)

We generalize the basic Moore machine by allowing each state to be associated with an action. Moreover, we define this action as an entry action that is carried out upon entering the state. Once we decide to simulate the FSM as a Moore-type machine, **FSMMoore** stars are needed to represent the states in the FSM. **FSMMoore** is a class derived from **FSMStateStar**.

4.2.2 Mealy-type state (FSMMealy)

An action is associated with a transition (arc) from one state to another in a Mealy machine. In a sense, those actions associated with transitions which come out of the same state may be viewed as *attached* to that state, as shown in Figure 14. The reason to do this is that we can just define a **FSMMealy** star which is derived from **FSMStateStar**. A **FSMMealy** state may have more than one action attached to it depending on the number of possible transitions. Nonetheless, each action is still supposed to be considered as associated with the transition.

4.2.3 Data Input (FSMDataIn)

The input data (or controls) are important for an FSM. Both actions and transitions rely on them. We define a type of **FSMDataIn** star to serve as an input interface. The scheduler uses its input portholes to receive the input data (or controls).

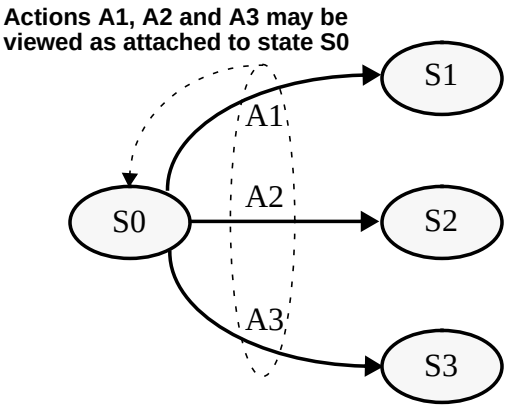


Figure 14. Actions may be viewed as attached to a state in a Mealy machine.

4.2.4 Data Output (FSMDataOut)

Sometimes, an FSM may need to send output to the outside world. Similar to **FSMDataIn** star, there is a type of **FSMDataOut** star in this domain to serve as an output interface for the FSM.

4.3 Interaction Semantics

From the point of view of implementation, a stand-alone FSM domain in Ptolemy is not very interesting, because in most of applications, the signal processing part is not negligible. Moreover, there are various dataflow models of Ptolemy domains. With the FSM domain mixed with them, we can get a much more powerful FSM model. Therefore, one of the main issues we would like to explore is how the FSM domain is combined with Ptolemy dataflow domains. We will focus on how FSM interacts with SDF in this paper.

4.3.1 FSM inside Dataflow

The inner FSM subsystem must externally behave like a dataflow actor to obey dataflow semantics. SDF actors generally consume and produce a fixed number of tokens on every input and every output. In Ptolemy, due to the current software implementation, a subsystem inside the SDF domain is constrained to behave like a *homogeneous* SDF actor, which consumes and produces a single token on each input and each output at each firing. Nevertheless, this constraint can be relaxed by modifying the wormhole mechanism in the SDF domain in Ptolemy.

We relate *one firing* of an SDF actor to *one state transition* of an FSM module. This means that when the FSM subsystem inside the SDF actor fires once by the outside SDF system, it reacts with exactly one state transition, as shown in Figure 15.

4.3.2 Dataflow inside FSM

Each action in an FSM can be specified by a subsystem in any domain. If the subsystem is an SDF graph, how much work should the dataflow scheduler do before returning control to the FSM scheduler? In *one state transition* of the FSM, there is exactly one action to be performed. Once this action (an SDF subsystem) is invoked by the FSM, the SDF scheduler should run through *one iteration*, consuming the input and producing the output.

If the SDF subsystem is a multirate system [2], which may consume and produce multiple tokens on the input and the output at each firing, the behavior becomes more subtle. One possibility is that when the input tokens are not sufficient to cycle through one SDF iteration, the SDF subsystem will simply return and produce no output tokens. Only when enough input tokens have accumulated will one SDF iteration be executed and the output tokens be produced. However, this is not always the most efficient approach.

4.3.3 Dataflow inside FSM inside Dataflow

Figure 16 shows an SDF subsystem embedded in an FSM that is inside another SDF domain. When the SDF subsystem needs to consume a large number of input tokens, for example a sub-

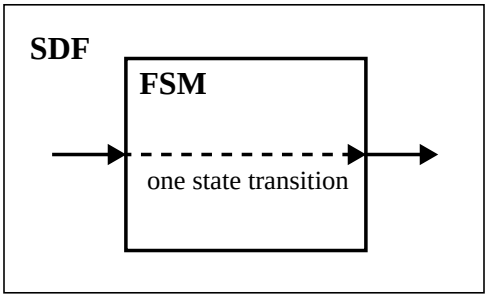


Figure 15. The FSM subsystem reacts to the firing with exactly one state transition.

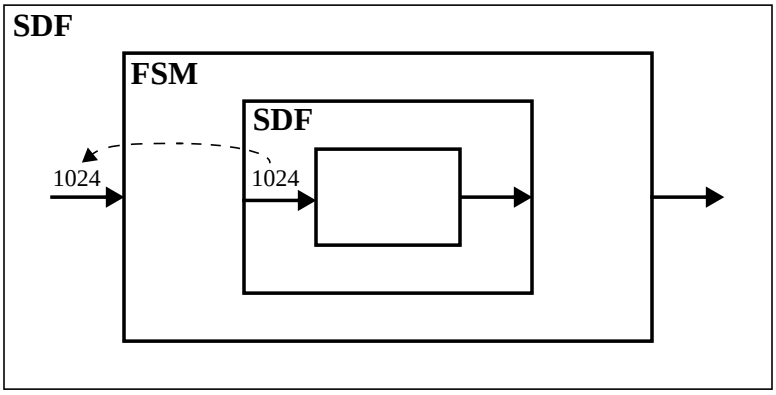


Figure 16. If an SDF subsystem is embedded in an FSM which is inside another SDF domain, the FSM will inform the outer SDF domain how many tokens are consumed at the input of the inner SDF subsystem.

system that computes a fast Fourier transform (FFT) consuming 1024 samples at its input, the method mentioned in Section 4.3.2 would be time-consuming because the FSM and inner SDF subsystems are invoked 1024 times before the inner SDF subsystem is actually ready to run through one iteration. A more reasonable alternative for this case is that the FSM will inform the outer SDF domain how many tokens are consumed at the input of the inner SDF subsystem, and then the FSM will not be invoked until there are sufficient input tokens for the SDF subsystem to cycle through one SDF iteration. Moreover, there may be more than one SDF subsystem associated with the FSM, it is required that all SDF subsystems consume the same number of input tokens to fire.

4.4 Graphical User Interface (GUI)

An FSM can be described by a visual syntax, such as the state transition diagram. The old and outdated visual interface to Ptolemy, called VEM, is not suitable for drawing the familiar bubble-and-arc diagrams for an FSM. A new visual editor (see Figure 17) is under development based on Tycho, a hierarchical syntax manager, which is part of the Ptolemy project. In addition to drawing the bubble-and-arc graph, users can click on a state or an arc to create or view a subsystem graph that is associated with it. After drawing the state transition diagram, users can further make an icon compatible with VEM and simulate it in Ptolemy.

Sometimes, no computation needs to be done in an action, and the outputs of the FSM are only fixed values. In Figure 1, for example, both FSMs just send zero or one to their outputs. For this case, the new editor also provides a way to specify the output values directly.

With the new visual editor, users can seamlessly traverse a hierarchical design that combines FSMs with dataflow block diagrams.

5.0 Application example

As mentioned in Section 2.2.2, Figure 2 is an HFSM representation of a three-bit counter with initialization and interruption mechanisms. In this section, we will detail it as a demonstration of an application that mixes the FSM and SDF domains in Ptolemy.

5.1 System Description

At the top level of the HFSM, there are two states: **Not Counting** and **Counting**. The **Counting** state is refined into three other automata, each of them a one-bit counter. The first one-bit counter, reacting to signal **a**, triggers the second one by emitting signal **b** for every two **a**'s. The second one reacts to signal **b** and emits signal **c** in every two **b**'s to trigger the third one. The third one outputs **end** reacting to every two **c**'s. The three automata of the one-bit counters are reacting *concurrently* to external signal **a**, i.e. the communication between them is considered as occurring in *one state transition* in the global behavior.

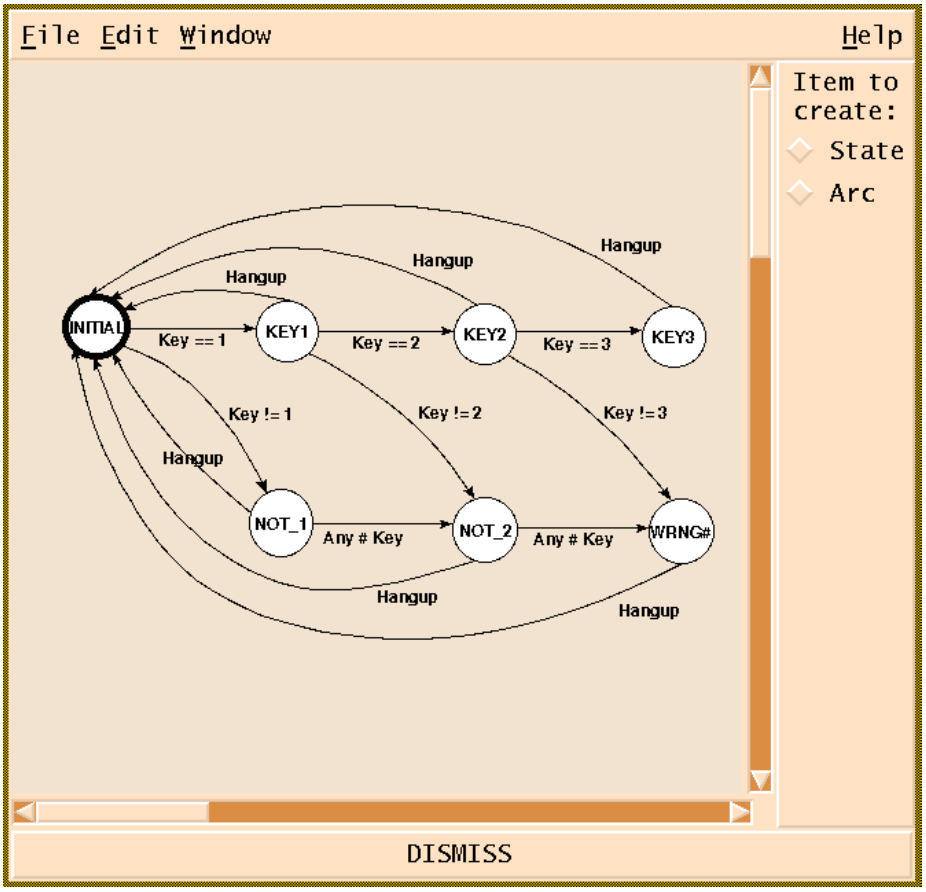


Figure 17. A new graphical editor developed in Tycho. (The drawing mechanisms of this editor have been mainly developed by Wan-teh Chang.)

Once **start** occurs, the automaton enters the **Counting** state. Then, three one-bit counters, which are initialized in states (**A0**, **B0**, **C0**), become active to react to external input **a**. After seven occurrences of **a,** the refined automata are in states **(A1**, **B1**, **C1**), and they output **end** reacting to one more occurrence of **a**.

Whenever **stop** occurs, the automaton stops the counting and enters the **Not Counting** state. Afterwards, three refined automata do not react to the input **a**. They will become active only when **start** occurs again.

5.2 Simulation

By using the SDF and FSM domains, we can describe and simulate the previous system in Ptolemy. In the SDF domain, a star must have tokens on all inputs and produce tokens on all outputs whenever it fires. Therefore, we have to use **1**/**0** value of a token to denote whether a signal event occurs or not. **1** means the signal is present, and **0** means it is not. This idea is important throughout the example.

The topmost level of the system is a dataflow model in the SDF domain, as shown in Figure 18. The built-in **TkButton** star, from SDF library, is used to generate three buttons labeled **Start**, **Stop** and **Count**, and also three outputs **start**, **stop** and **count**, respectively. The output value is 0.0 unless the corresponding button is pushed. When the **Start** button is pressed, for example, the output values are 1.0, 0.0 and 0.0, respectively. If no button is pushed, this star sends 0.0 on all outputs. The **3bitCounter_FSM** block is a subsystem in the FSM domain, and we will discuss it below. Finally, another built-in **TkBreakPt** star is used to catch the output of the FSM. When its

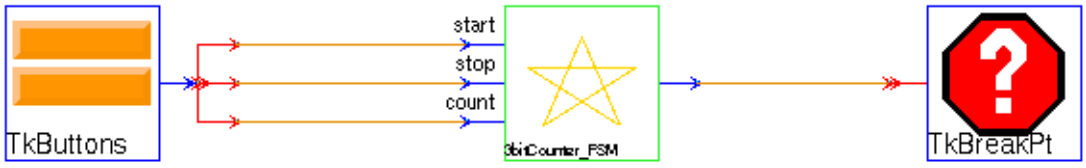


Figure 18. System diagram in SDF domain for simulating the 3-bit counter.

input value becomes 1.0, i.e. there is a signal occurring, it issues message and pauses the simulation.

As mentioned earlier, the **3bitCounter_FSM** block is an FSM subsystem. We use a Moore-type machine as a demonstration to describe this subsystem (a Mealy-type machine can be used in a similar way). It consists of two states, **Not Counting** and **Counting**, as shown in Figure 19. The bold circle around the **Not Counting** state means that this state is the initial state of the FSM. In the state **Not Counting**, no computation needs to be done, and the **3bitCounter_FSM** subsystem only sends zero to the output.

The SDF graph associated with the **Counting** state is more subtle, as shown in Figure 20. There are three FSM subsystems (**1bitCounter_FSM** blocks) in the graph, each a one-bit counter.

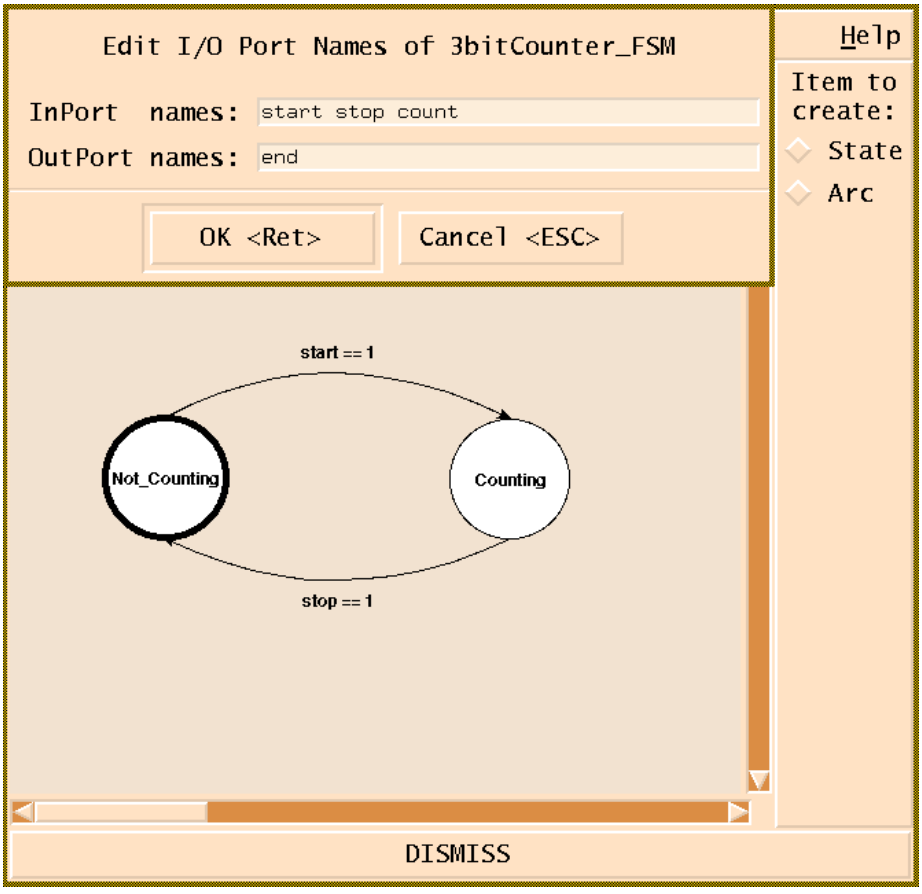


Figure 19. The FSM of the 3-bit counter.

Every one-bit counter is connected with the external **start** signal which serves as an initialization signal and starts every counter in its initial state. The external **stop** signal is not necessary in this level, so we just discard it. The external **count** signal is connected to the first one-bit counter as its input signal, and is like the **a** signal in Figure 2. The output signal **end** of the third one is sent to the outer domain. The three one-bit counters are put in a sequential connection. However, they all fire in one SDF iteration, so they are considered *concurrent* in that they all fire once per state transition of the outer FSM.

The FSM subsystems of the three one-bit counters are all the same, as shown in Figure 21, and each of them simply consists of two states, **State0** and **State1**. The bold circle around the **State0** state means that this state is the initial state. A simple SDF galaxy (see Figure 22) associated with the **State0** state will do the desired job. This galaxy discards the **reset** signal and reacts to the **input** signal. Only when the **input** value is one will the **output** value be one to represent that a signal occurs in this state. In the **State1** state, the FSM just sends zero to output, i.e. no **output** signal occurs in this state.

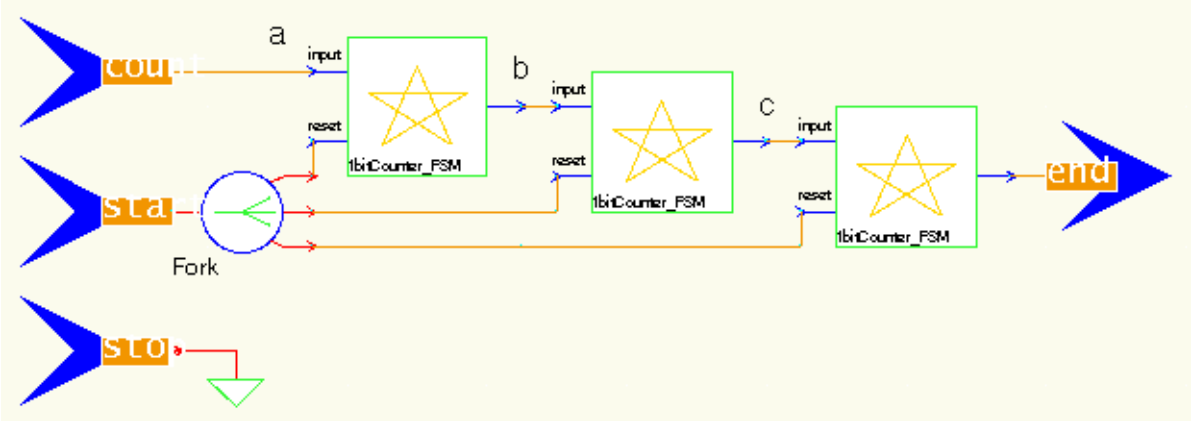


Figure 20. The SDF galaxy associated with the Counting state in the 3-bit counter.

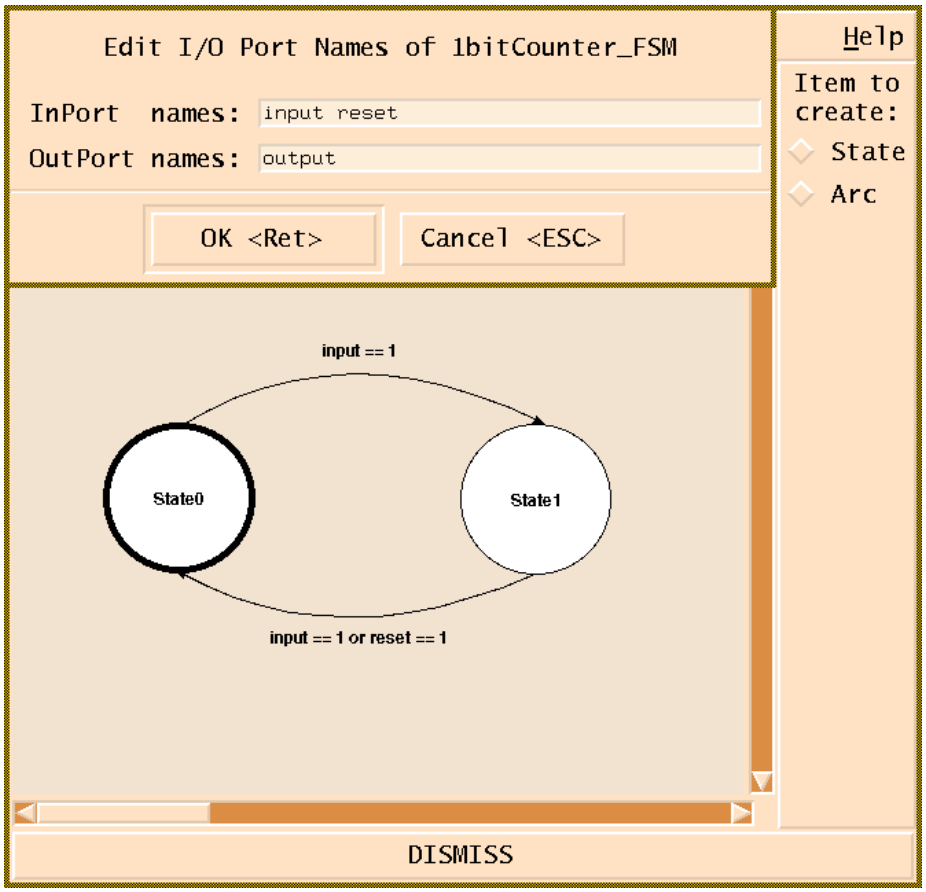


Figure 21. The FSM of the 1-bit counter.

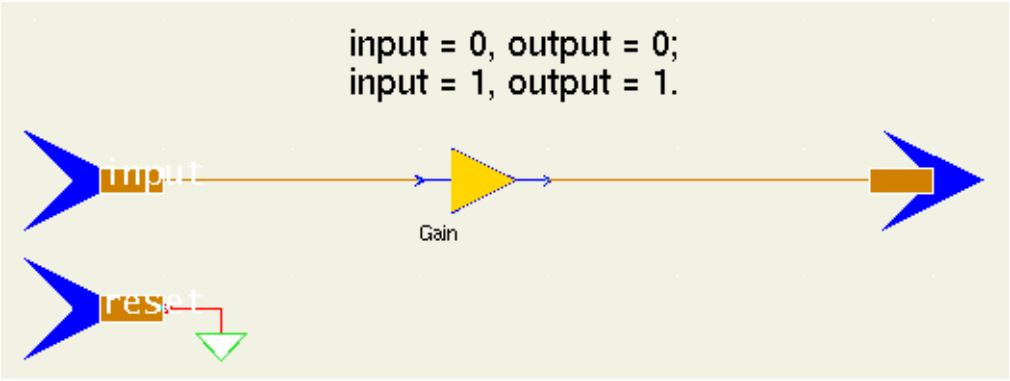


Figure 22. The SDF galaxy associated with the State0 state in the 1-bit counter.

5.3 Discussion

This example demonstrates how FSM and SDF domains are hierarchically nested to specify and simulate a system in Ptolemy. The FSM models mainly focus on describing the control flow of the system, and the SDF graphs are for the numeric-computation of the system.

As shown in Figure 20, the SDF galaxy, in which three one-bit counters are interconnected, encapsulates the semantic property of *concurrency.*

Look at the FSM of the one-bit counter as shown in Figure 21. When the current state is **State0** and the **input** value is zero, i.e. no **input** signal is present, the FSM remains in **State0**. Then, the SDF galaxy associated with **State0**, as shown in Figure 22, sends zero as its output. In another situation, when the current state is **State1** and the **input** value is one, the FSM makes a transition to **State0**, and the galaxy sends one as its output. In the state **State0**, the output values may be different based on different transitions. This is because we generalize the Moore machine by associating an action with each state, and the action may output different values in different situations. Therefore, our Moore-type machine does not have the constraint that each different output value requires a different state, and this can reduce the complexity of the state space.

In addition to the example illustrated above, other complicated applications can be developed by similar steps. For example, the answering machine mentioned in Section 1.0 contains both signal processing and sophisticated control. The key principle is to use the dataflow model to specify the signal processing part and the FSM model to specify the control flow part. Furthermore, the dataflow model can be used to describe the concurrency semantics of the system.

6.0 Conclusion

In this paper, we introduce an FSM domain into Ptolemy. By hierarchically nesting two distinct domains, FSM and dataflow, we can describe HFSMs with desired semantic properties: FSM, concurrency and hierarchy. Most importantly, we do not need a new complicated semantics for a complex HFSM, we can use basic FSM models mixed with dataflow graphs to achieve the goal.

Furthermore, FSM semantics is useful for control-oriented systems, and dataflow semantics for numeric-intensive systems. With the two models brought together, a system doing both sophisticated control and signal processing can be specified and simulated in Ptolemy.

The interaction semantics defines the interaction between different semantic models. We define *one state transition* in the FSM domain, and relate it to a firing in the SDF domain.

Although we mainly discuss mixing FSM with SDF in this paper, there are many other (even non-dataflow, such as *discrete-event*) domains in Ptolemy which can interact with FSM domain by similar mechanisms.

7.0 Future Work

The current implementation of the FSM domain is a simulation domain in Ptolemy. We may extend its capabilities with the code generation in C, C++, Tcl, and VHDL.

A visual syntax using a graphical editor describes the FSM applications in this paper. A language with textual syntax may also describe the FSM automata, such as Esterel [1]. We may extend the FSM domain to allow using the textual syntax as the FSM description.

For further implementation, we may consider FSM optimization in this domain using some FSM minimization tool, like SIS [8].

8.0 References

[1] F. Boussinot, R. De Simone, “The ESTEREL Language,” *Proceedings of the IEEE*, vol. 79, no. 9, September 1991.

[2] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, “Multirate Signal Processing in Ptolemy,” *Proc. of the Int. Conf. on Acoustics, Speech, and Signal Processing*, Toronto, Canada, April 1991.

[3] 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.

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

[5] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, M. Trakhtenbrot, “STATEMATE: A Working Environment for the Development of Complex Reactive Systems,” *IEEE Tr. on Software Engineering*, vol. 16, no.4, April 1990.

[6] F. Maraninchi, “The Argos Language: Graphical Representation of Automata and Description of Reactive Systems,” *Proceedings of the IEEE Workshop on Visual Languages*, Kobe, Japan, October 1991.

[7] E. A. Lee and T. M. Parks, “Dataflow Process Networks,” *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773-801, May, 1995.

[8] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, et al., “Sequential circuit design using synthesis and optimization,” in *Proc. of ICCD* (International Conference on Computer Design), Cambridge, MA, USA, 11-14 Oct. 1992). Los Alamitos, CA, pp. 328-33, 1992.

[9] J. Rasure and C. S. Williams, “An Integrated Visual Language and Software Development Environment,” *Journal of Visual Languages and Computing*, vol. 2, pp. 217-246, 1991.

[10] M. von der Beeck, “A Comparison of Statecharts Variants,” in *Proc. of Formal Techniques in Real Time and Fault Tolerant Systems*, LNCS 863, pp. 128-148, Sprinter-Verlag, Berlin, 1994.