COMMUNICATION SYSTEMS MODELING IN PTOLEMY II

by

Ye Zhou

Memorandum No. UCB/ERL M03/53

18 December 2003

COMMUNICATION SYSTEMS MODELING IN PTOLEMY II

by

Ye Zhou

Memorandum No. UCB/ERL M03/53

18 December 2003

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

Synchronous Dataflow (SDF) is useful in modeling communications and signal processing systems. We describe a communication actor library based on the SDF semantics in Ptolemy II and show how to use these actors to simulate communication systems. However, many communication and signal processing systems nowadays use adaptive algorithms and control protocols. These sometimes violate SDF principles in that SDF actors must have fixed rates during execution. A. Girault, B. Lee, and E. A. Lee proposed in [7] a new model of computation called Heterochronous Dataflow (HDF). HDF extends SDF by allowing rate changes in actors during execution. HDF is a heterogeneous composition of SDF and Finite State Machines (FSM). We describe an HDF domain implementation in Ptolemy II. Examples and discussions are given to show how HDF can be used in various forms.

Acknowledgements

I would like to thank my research advisor, Professor Edward A. Lee, for introducing me to this research area. This work would not be possible without his consistent support, guidance, and his trust and encouragement.

I would also like to thank Professor Alberto Sangiovanni-Vincentelli for kindly agreeing to serve as the second reader of this report.

I wish to thank all members in the Ptolemy group for the friendly and positive atmosphere. In particular, Xiaojun Liu has given me constant help since I started learning to use and write Ptolemy softwares. Steve Neuendorffer offered useful suggestions and discussions when I was implementing HDF.

I am grateful to all my friends, for their help and care, the joy they bring, and their tolerance to me :)

Finally, I would like to thank my parents for their consistent support and encouragement. This report is dedicated to them.

Contents

1. Introduction

1.1 Ptolemy II

The Ptolemy Project at the University of California, Berkeley studies heterogeneous modeling, simulation and design of concurrent systems. The current software infrastructure is called Ptolemy II, and is constructed in Java. It is a system-level design environment that supports component-based hierarchical and heterogeneous modeling.

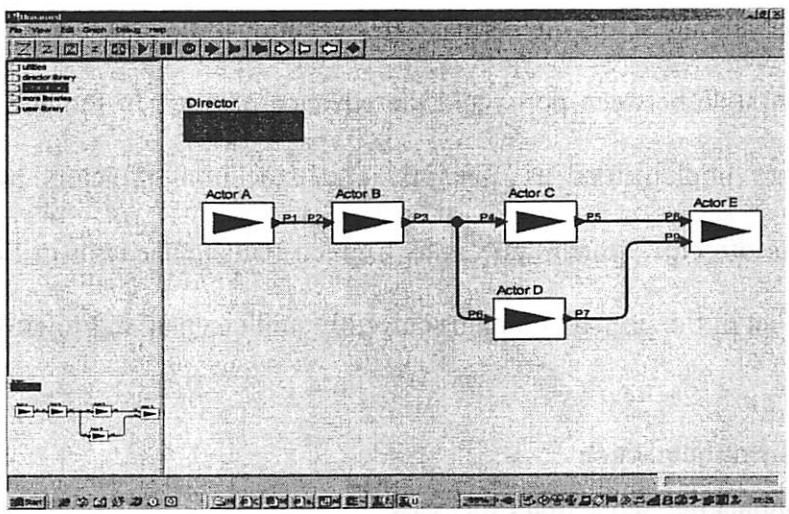


Figure 1.1 Vergil: the graphical user interface of Ptolemy II

In Ptolemy II, models can be constructed graphically using a graphical user interface (GUI) called Vergil, as shown in Figure 1.1. Each block is called an *actor*, which represents a function or a set of functions that map the state and inputs to outputs. An atomic execution of an actor is called a *firing*. An actor receives data at its *input ports* and produces the result at its *output ports*. Actors that only have output ports are called *source* actors. Actors that only have input ports are called *sink* actors. Ports can be connected to

set up communication channels among actors. Data samples (or *tokens*) are sent over channels connecting an output port to an input port. For example, in Figure 1.1, the source actor *A* produces tokens at port *P*1. Actor *B* receives these tokens at its input port *P*2, performs its computation, and sends the result to its output port *P*3.

1.2 Models of Computation

A model of computation (MOC) provides a set of rules that specifies the execution and communication among actors. It defines the execution order (schedule) of actors, the communication style between ports, and the advance of time. In Ptolemy II, models of computation are implemented as *domains*. The execution of actors in a domain is managed by its *director*, which appears as a green rectangular icon in Figure 1.1. The following is a list of the primary domains currently implemented in Ptolemy II:

CI Component Interaction
CSP Communicating Sequential Processes
CT Continuous Time
DE Discrete Events
DDE Distributed Discrete Events
DT Discrete Time
FSM Finite State Machines
PN Process Networks
SDF Synchronous Dataflow
SR Synchronous Reactive
TM Timed Multitasking

In the next section, we will discuss SDF in detail. Later, we will also discuss a newly implemented model of computation called heterochronous dataflow (HDF). HDF is originally proposed by A. Girault, B. Lee, and E. A. Lee in [7]. It extends SDF by combining SDF with FSM.

The Ptolemy II software provides a variety of actor libraries, which include math functions, logic controls, signal processing actors, etc. Some actors are *domain specific*, which means they may only work under certain domains. For example, the "Integrator" actor works only in the CT domain. In Ptolemy II, efforts are being made to design *domain polymorphic* actors so they can be reused. The "AddSubtract" actor and "Scale" actor are examples of this.

1.3 Hierarchy and Heterogeneity

Ptolemy II uses *composite actors* to construct models in a hierarchy. A composite actor is an actor that contains other actors. In contrast, *atomic actors* are actors that cannot contain other actors and only appear at the bottom of the hierarchy. As shown in Figure 1.2, actor *B* is a composite actor that contains *B*1 and *B*2. Actors *A* and *C*, as well as *B*1 and *B*2 are all atomic actors. The outside and inside actors interact with each other by passing tokens through the ports of the composite actor. A composite actor can be contained by another composite actor, so the depth of hierarchy is arbitrary.

A composite actor may have two directors. The outside director (such as $D1$ in Figure 1.2) is called the *executive director*, which views the composite actor as a black box and manages its execution. In Ptolemy II, a composite actor is called *opaque* if it has a director inside. The inside director is called the *local director*, and manages the execution of the inside actors. The executive director and local director do not have to be same, so heterogeneous modeling is implemented. A composite actor without a local director is called a *transparent* composite actor. Actors inside a transparent composite actor follow the semantics of the executive director.

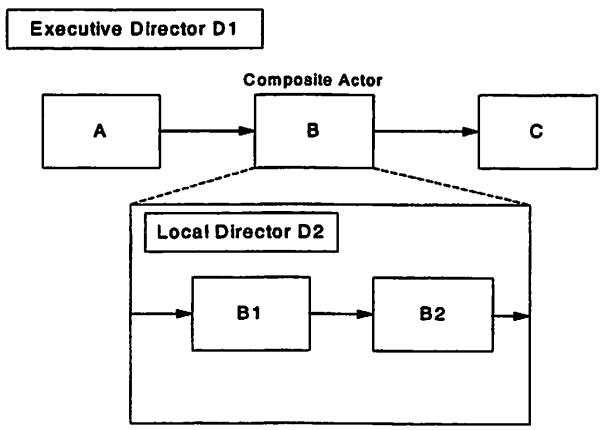


Figure 1.2 Hierarchical model using an opaque composite actor

2. Communications System Modeling

2.1 Synchronous Dataflow (SDF)

Synchronous dataflow (SDF) is a well-understood model of computation in Ptolemy II. It is useful in design and modeling of multi-rate signal processing algorithms. Figure 2.1 shows a simple SDF model. Under the SDF model of computation, actors consume and produce a fixed number of tokens on each firing. These numbers are often referred to as *consumption rates* and *production rates*, and together give the *rate signatures* of the actors. In our notation, we use integers in parentheses adjacent to the ports to indicate these port rates, as shown in Figure 2.1. The channels that connect SDF actors represent streams of tokens, and can be viewed as first-in-first-out (FIFO) queues.

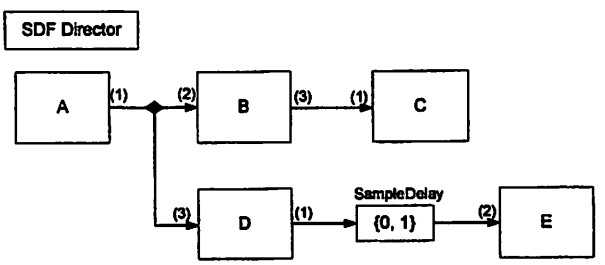


Figure 2.1 A simple SDF model

A channel may have *delay*. A *unit of delay* in a channel from actor A to B means that the n-th token consumed by B will be the $(n-1)$ -th token produced by A. This is implemented as an initial token in the channel. Ptolemy II uses the SampleDelay actor to produce the initial tokens, specified by its *initialOutputs* parameter. The number of initial tokens is called the *initial production rate* of the actor. For example, in Figure 2.1, a

SampleDelay actor is inserted between actor D and E. The *initialOutputs* parameter are set to be {0, 1}, indicating there are two units of delay. For most actors other than SampleDelay, the initial production rate is zero.

An *iteration* of SDF is the minimal set of firings that returns the queues to their original sizes. An SDF *schedule* describes the numbers of firings and execution order of the actors in an iteration. A necessary condition for the existence of an SDF schedule is the existence of solution to the *balance equations* for each channel [12] [13]. The balance equation for a channel that connects port P_i of actor A to port P_j of actor B is defined as:

$$r_i f_A = r_j f_B$$

where r_i and r_j are the port rates of P_i and P_j respectively; f_A and f_B are the numbers of firings for actor A and B respectively. If an SDF model has N actors and M channels in total, then there will be M such equations and N unknowns. The unknowns are the number of firings f_i for each actor i in one iteration. It is shown that for such a system, there either exists a unique minimal positive solution, or the only solution is all f_i's are zero [12] [13]. An SDF graph is *inconsistent* if there is no positive solution to the balance equations.

Consistency is not a sufficient condition to determine a schedule. In the example in Figure 2.2(a), the SDF model deadlocks because there is a loop dependency between the two actors. The problem is resolved in Figure 2.2(b) by introducing a unit delay in the loop. Actor B can fire given the initial token produced by the SampleDelay, and then

actor *A* can fire, etc. This suggests that whether a schedule exists for an SDF graph is also dependent on how actors are connected and delays in the channels. Lee and Messerschmitt have developed efficient algorithms in [12] to determine the existence an SDF schedule and how to compute it statically. An SDF schedule is deadlock-free, and bounded in schedule length and memory usage.

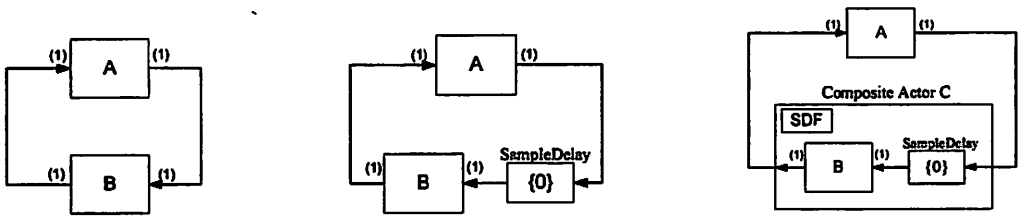


Figure 2.2 (a) Loop dependency results in deadlock. (b) Deadlock resolved by introducing a unit delay in the loop. (c) Deadlock introduced by composition.

Hierarchical SDF

There are two ways to view a hierarchical SDF model. First, the model can be flattened to remove the hierarchy. In Ptolemy II, a model containing transparent composite actors is scheduled in this way. Secondly, if the composite actor is opaque, it can be viewed as an atomic actor from the executive director. But a composite actor hides relations among actors, and the SDF graph may deadlock. For example, Figure 2.2(c) shows the same model as in Figure 2.2(b), but actor *B* and the SampleDelay actor is embedded in an opaque composite actor *C*. The composite actor hides the initial token from the higher hierarchy, causing a deadlock to occur.

Composing actors in a model that does not have SampleDelay actors may also introduce deadlock. Below is the example used in [17]. In Figure 2.3(a), port *P*1 and *P*2 can consume tokens sequentially, and the model is deadlock-free. In Figure 2.3(b), a new directed loop is introduced by composing actors *A* and *B*, and the model deadlocks.

(a) (b)

Figure 2.3 Deadlock introduced by composing actor *A* and *B* into one composite actor.

SDF with Other Domains

Generally, an SDF model can be heterogeneously composed with other domains. However, SDF models often require multiple input tokens per firing, while most other domains such as CT operate on single token per firing. If a composite actor of another domain is embedded in SDF, it must behave like an SDF actor. That is, it must have fixed consumption and production rates. If an SDF composite actor is embedded in other domains, it will refuse to fire until there are enough tokens available at all the inputs.

2.2 Communications Actors

Many signal processing actors can be implemented based on the SDF semantics. These actors include filtering, spectrum analysis, etc. We have recently added a communications sublibrary to the signal processing library. We use Boolean tokens to represent the binary bits in communication actors. Here we give a brief introduction to the current actors to see how they are designed and used in the SDF context.

HadamardCode

This is a source actor that produces a Hadamard (Walsh) codeword by selecting a row from a Hadamard matrix. A Hadamard matrix of order n is defined in the following way:

$$H_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad H_{n+1} = \begin{bmatrix} H_n & H_n \\ H_n & -H_n \end{bmatrix}$$

Therefore, H_n is a 2^n by 2^n square matrix. Each row or column of the Hadamard matrix is orthogonal to the others. For this reason, a Hadamard code is often used to generate orthogonal codes. The order n is specified by the *log2Length* parameter and the length of the codeword is 2^n. The row index is specified by the *index* parameter or the associated *index* port. On each firing, the actor produces one Boolean token of the codeword in sequence. We did not design the actor to produce the whole codeword in one firing because the production rate 2^n would grow exponentially with the order n. This could make the schedule fairly large if the actor is used in SDF.

LineCoder

The LineCoder actor is written by Prof. Lee and Steve Neuendorffer. The LineCoder consumes k Boolean tokens on each firing, where k is specified by the *wordLength* parameter. The *table* parameter is an array of length at least 2^k. The k Boolean inputs are taken to be a binary digit that indexes the array, where the first input is taken to be the low-order bit of the index. The output is the corresponding element in the *table*.

The SDF schedule, if it exists, ensures that the LineCoder receives tokens in multiple of k, so that encoding can be carried out successfully. Another way to design the LineCoder is to implement it in cyclo-static dataflow (CSDF), where the consumption rates and production rates of an actor vary from one firing to the next in a cyclic pattern [4] [17]. A LineCoder in CSDF has a *cycle* of k firings. It consumes 1 token on each firing. Since the output cannot be produced until there are k inputs available, the actor does not produce any token on the first $(k-1)$ firings and produces 1 token (the encoded result) on the k-th firing. Figure 2.4(a) and (b) shows the different forms of LineCoder in SDF and CSDF when $k = 3$. The vector $(r_1, r_2, ..., r_k)$ adjacent to a port indicates the port's rate of each firing in a cycle.

(a) LineCoder in SDF (b) LineCoder in CSDF

Figure 2.4

CSDF has some advantages over SDF. CSDF offers the opportunities to eliminate unnecessary computation (dead code) and explore additional parallelism [17]. T. M. Parks et al. also showed in [17] how to reduce the number of actor firings to be scheduled by transforming CSDF graph into SDF graph. But such transformation may introduce deadlock, or lose the advantages of CSDF such as dead code elimination.

Slicer

The Slicer functions as a decoder for the LineCoder. The *table* and *wordLength* parameters have the same meaning as in LineCoder, but the table is only constrained to be an array of complex numbers. On each firing, the Slicer consumes one complex token and produces k Booleans, where k is specified by the *wordLength* parameter. It computes the Euclidean distance between the input data and the elements in the table. The values of the output Booleans correspond to the index of the element that minimizes the distance, encoded in binary, with the lower-order bit coming out first.

Scrambler

This actor scrambles the input Boolean sequence using a feedback shift register. The consumption rate and production rate are both 1. The initial state of the shift register is given by the *initialState* parameter. The taps of the feedback shift register are given by the *polynomial* parameter, which should be a positive integer. The n-th bit of this integer indicates whether the n-th tap of the delay line is fed back. All the bits that are fed back are exclusive-ored together (i.e., their parity is computed), and then exclusive-ored with the input bit. The result is produced at the output and shifted into the delay line.

Figure 2.5 shows an example of the scrambler. The polynomial is "11001" in binary, or "031" in octal. With a proper choice of polynomial, the resulting output appears highly random even if the input is highly non-random. The Scrambler actor is often used in two ways. First, it adds randomness to the input sequence. This improves the accuracy of timing recovery by eliminating long sequences of zeros. Second, the actor can be used to generate a pseudo-random code, which is a nearly orthogonal code.

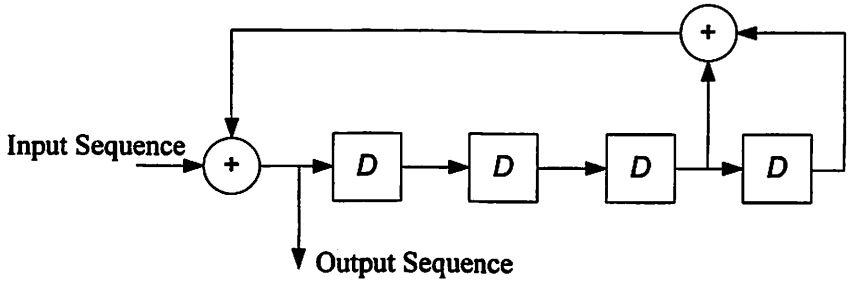


Figure 2.5 A scrambler example

Descrambler

This actor descrambles the input bit sequence using a feedback shift register. It also has an *initialState* parameter and a *polynomial* parameter, which have the same meaning as in the Scrambler actor. This is a self-synchronizing descrambler that will exactly reverse the operation of the Scrambler if the parameters are the same.

HammingCoder

This actor encodes the input symbols into a Hamming codeword. Generally, an encoder that encodes k bits into a codeword of length n is often called an (n, k) encoder. It can be

implemented as an actor with consumption rate k and production rate n. Similar to the LineCoder actor, an (n, k) encoder could also be implemented in CSDF with a firing cycle k. The n output tokens are produced on the k-th firing.

The Hamming code is a typical example of a linear block code. For an (n, k) linear block code, the k inputs and n outputs are viewed as two row vectors. The codeword is generated by multiplying the input with a specified *generator matrix* G, which consists only of zeros and ones. The dimension of G is k by n. The addition that occurs in the matrix multiplication is modulo-two [14] [16]. The n and k of a Hamming code must satisfy the following:

$$n = 2^m - 1$$
$$k = 2^m - 1 - m$$

for some integer m, where m is called the order of the Hamming code. The lowest order is $m = 2$, and $(n, k) = (3, 1)$. The most commonly used Hamming code is the (7, 4) Hamming code, where $m = 3$. The generator matrix for the (7, 4) Hamming code implemented in the HammingCoder actor is given by:

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

HammingDecoder

This actor is the decoder of the HammingCoder actor. For an (n, k) Hamming decoder, the consumption rate is n and production rate is k.

ConvolutionalCoder

The ConvolutionalCoder actor encodes an input sequence using a convolutional code. The input sequence enters a shift register. The initial state of the shift register is given by the *initialState* parameter. Similar to the Scrambler actor, the *polynomialArray* parameter is an array of positive integers. Each integer indicates one polynomial used for computing the parities. The results are sent to the output port. For an (n, k) convolutional code, k is the number of bits per firing that are shifted into the shift register. It is specified by the *uncodedRate* parameter. The production rate n is indicated by the number of integers in the *polynomialArray*. For example, in Figure 2.6, $k = 1$; the *polynomialArray* is {101, 111} in binary, or {05, 07} in octal, and therefore $n = 2$.

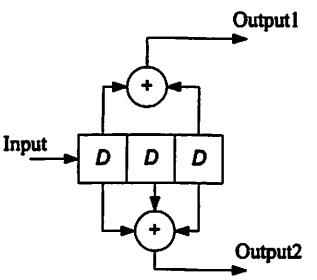


Figure 2.6 A 1/2 convolutional code

ViterbiDecoder

The Viterbi algorithm is an optimal way to decode a convolutional code. The convolutional code is specified by the *uncodedRate* parameter and the *polynomialArray* parameter, same as in the "ConvolutionalCoder" actor. The decoder finds the most likely data sequence given noisy inputs by searching all possibilities and computing the distance

between the codewords they produce and the observed noisy data. The sequence yielding the minimum distance is the decoded output.

The decoder provides two modes of decoding, namely soft decoding and hard decoding, specified by the Boolean *softDecoding* parameter. For hard decoding, the input port accepts Boolean tokens and computes the Hamming distance. For soft decoding, the ViterbiDecoder handles binary antipodal constellations. The input port accepts double tokens and computes the Euclidean distance. The parameter *constellation* should be a double array of length 2 that specifies the amplitude of "false" and "true" inputs. In general, soft decoding has a lower probability of decoding error, but hard decoding is less computationally expensive. Moreover, hard decoding can be used when there is no direct observation of the noisy data, but only observations of a bit sequence that may have errors in it.

There is some delay between the reading of input data and the production of decoded output data. That delay, which is called the *trace-back depth* or *truncation depth* of the decoder, is controlled by the *delay* parameter. Larger values of *delay* generally reduce the probability of error. In order to guarantee the fixed rates of SDF actors, false-valued tokens will be produced during the first *delay* firings of this actor.

TrellisDecoder

The TrellisDecoder is a generalization of the ViterbiDecoder that handles non-binary antipodal constellations. The trellis code extends the convolutional code by mapping each

codeword into a complex number. This can be done by using the LineCoder actor. For an (n, k) convolutional code, the *wordLength* of the LineCoder should be set to n, and the size of the *table* should be 2^n. On each firing, the LineCoder consumes the n tokens produced by the ConvolutionalCoder and outputs the corresponding complex number according to its *table*.

The TrellisDecoder consumes one complex token on each firing. It finds the most likely data sequence by searching all possibilities and computing the Euclidean distance between the complex numbers they produce and the observed noisy data. The sequence yielding the minimum distance is the decoded output.

Like the ViterbiDecoder, the TrellisDecoder also has a *delay* parameter, and false-valued tokens will be produced during the first *delay* firings to guarantee the fixed rates in SDF actors.

2.3 Example

We use an example to illustrate how these actors can be used in SDF to simulate communication systems. The model shown in Figure 2.7 compares the coded and uncoded transmissions of a 4PSK signal through a complex Gaussian noise channel. The 4PSK signal is generated by using a LineCoder actor where the constellation is defined as in Table 2.1. The uncoded channel uses Slicer to decode the 4PSK signal. The coded

channel uses a 1/2 rate convolutional code and a TrellisDecoder to decode the 4PSK signal.

Table 2.1 4PSK constellation
"T" stands for "true" input and "F" stands for "false" input

Input	FF	FT	TF	TT
Constellation	$1.0+i$	$-1.0+i$	$-1.0-i$	$1.0-i$

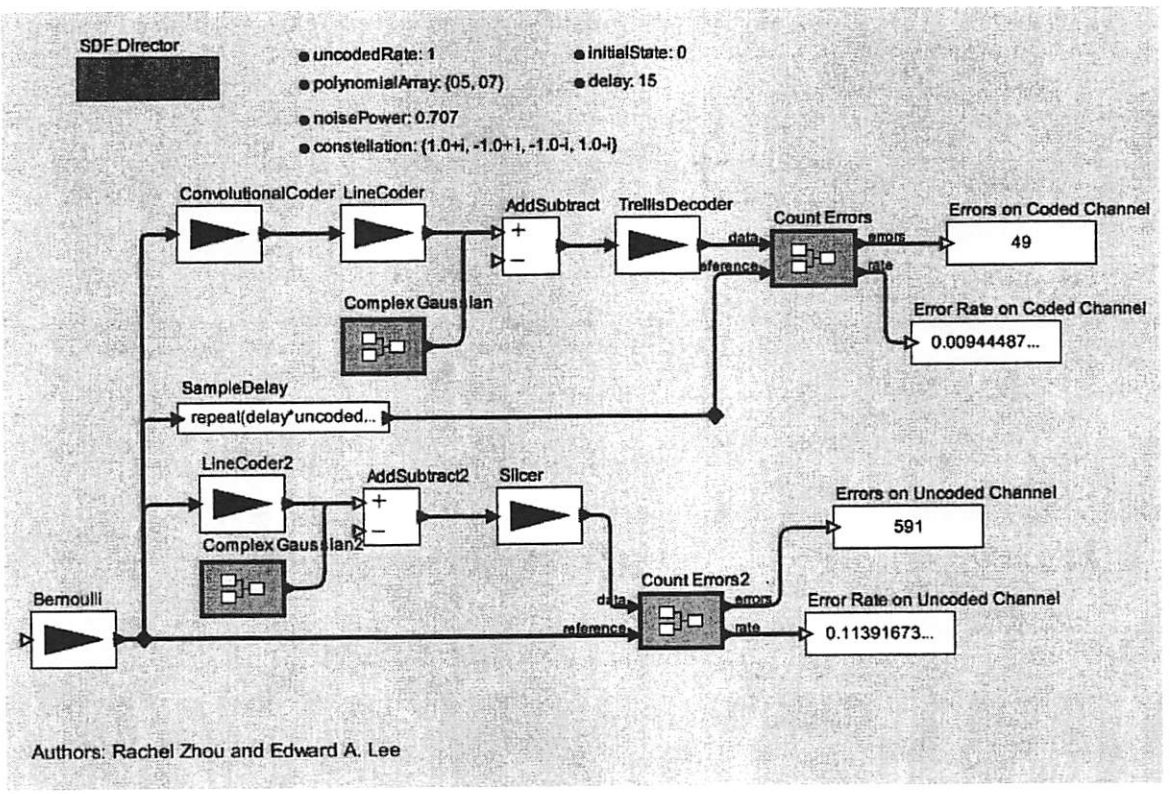


Figure 2.7 A comparison between coded and uncoded 4PSK signal transmission

The number of bit errors and error rate are displayed in the icons in Figure 2.7. When the noise power is 0.707, or equivalently the signal to noise ration (SNR) ≈ 2, or 3dB, we can see that the coded channel is more than 10 times better than the uncoded channel.

Similar examples can be constructed using other communications actors. Although communications and signal processing actors are not SDF-specific, they often require multiple tokens to fire. An SDF schedule, if it exists, guarantees that these actors always have sufficient data, and deadlock will never occur. Memory usage is also decidable prior to execution. This is why SDF is useful in communications and signal processing modeling and simulation.

3. Heterochronous Dataflow (HDF)

3.1 Motivation for HDF

Although SDF is very powerful in modeling signal processing systems, rate parameters are fixed and must be specified *a priori*. This is a necessary condition to ensure that an SDF schedule can be determined prior to execution. If any actor changes its port rates during execution, the numbers of tokens arriving at and leaving the FIFO queues are not balanced, or there may be insufficient delay to resolve deadlock. An error is then reported.

However, as the following examples shows, many signal processing systems nowadays need to implement control protocols and adaptive algorithms that require dataflow variation during execution. The constraint of fixed rates in SDF limits its applications to these kinds of systems.

- **Switch coding in a wireless channel**

  In wireless communication, channels are varying depending on weather, the surrounding objects, the distance between the transmitter and receiver, etc. It is sufficient to use a simple channel coding when the signal is strong. A more sophisticated coding scheme will be used when the signal is poor. Hence the data flow is switched between two encoders; each may have different consumption and production rates.

- **Pilot Training**

Channel equalization is often used in communication systems to reduce intersymbol interference (ISI). When the channel is unknown, the system needs to estimate it by implementing a *training mode* at the start-up of the execution. In this mode, the transmitter sends a *training signal* that is known to the receiver. This provides the initial estimation of the channel. Information data are sent after the training signal, and the receiver will make symbol decision based on the noisy signal. Further estimation can be performed using a decision-directed equalizer. Figure 3.1 shows the block diagram of an adaptive equalizer [14].

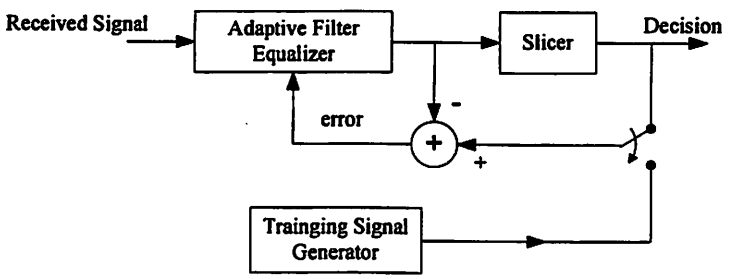


Figure 3.1 Block diagram of an adaptive equalizer.
This graph is partly excerpted from reference [14], page 518.

- **Data-dependent sampling**

Figure 3.2 shows the resampling part of the carrier recovery algorithm in the Ethereal Sting Project [18]. The model was originally built by Steve Neuendorffer. The rest of the system consists of regular SDF actors, and for clarity, is not shown here. In Figure 3.2, if the BooleanSwitch actor receives a "true" token at its *control* port, the input token (the baseband signal) is directed to its *trueOutput* port and further processed to

plot the estimated constellation. If the *control* port receives a "false" token, the input token is directed to the *falseOutput* and discarded from the system. Steve Neuendorffer uses the Process Network (PN) domain to implement the model because the behavior of BooleanSwitch violates SDF principles. We will discuss later an alternative solution for this example.

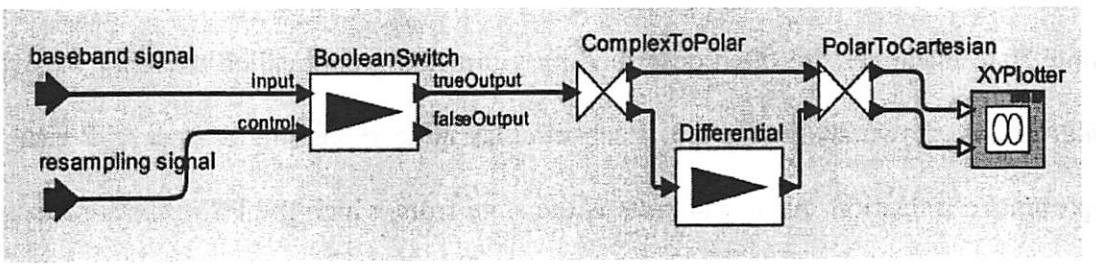


Figure 3.2 Data-dependent sampling

The three examples above use largely SDF actors, with some simple flow control in the model. Although "simple," the varying flow direction and rates prohibit us to use SDF. Other dataflow models such as dynamic dataflow (DDF) and Boolean dataflow (BDF) [5] [6] are not restricted to fixed rates. However, in DDF and BDF, many questions such as deadlock are undecidable. And it may seem that DDF and BDF are too general for these applications with simple control. It would be ideal if we could implement flow control mechanism while preserving the benefits of SDF. Then we would also be able to reuse all the existing actors in the signal processing libraries. A natural approach is to combine SDF with finite state machines (FSM). FSM is widely used in control systems. The changes in system dynamics are well defined by state transitions in FSM. This leads to a new model of computation called heterochronous dataflow (HDF), originally introduced by A. Girault, B. Lee and E. A. Lee in [7]. In the next subsection, we will give a brief

introduction to the basic FSM. The HDF model of computation extends the semantics of SDF and FSM while preserving backward compatibility.

3.2 Finite State Machines (FSM)

A finite state machine (FSM) consists of a set of *states* and *transitions* among states. Finite state machines are often described by a directed graph, called the *state transition diagram*, as shown in Figure 3.3. Each circular node represents a state and each arc represents a transition. An *initial state* is the state from which the FSM starts execution. Each transition is labeled with a *guard*, which is often a Boolean function of the input ports. A transition is enabled if the guard is true. For example, suppose the FSM in Figure 3.3 is currently in State *A*. The guard " $P > 0$ " suggests that a state transition to State *B* will occur if the token value at port *P* is greater than 0. A transition may also be associated with *actions* to produce outputs or set parameters.

Within the FSM domain of Ptolemy II, we only discuss *deterministic* and *reactive* FSM's. An FSM is deterministic if from any state there exists at most one enabled transition. An FSM is reactive if from any state there exists at least one enabled transition. In Ptolemy II, an error will be reported if multiple transitions are enabled. If no transition is enabled, Ptolemy II assumes that an implicit transition back to the current state is made.

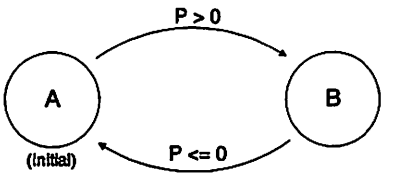
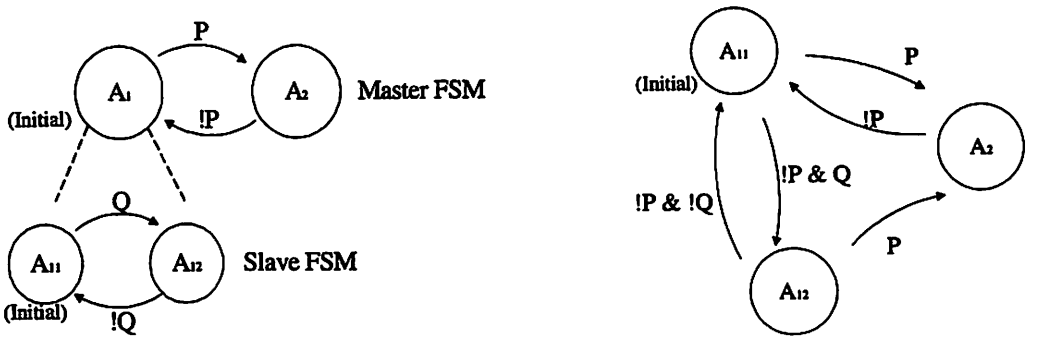


Figure 3.3 State transition diagram of FSM

Hierarchy

An FSM can be constructed hierarchically, where a state may be further refined into another FSM, as shown in Figure 3.4(a). The inside FSM is called the *slave FSM* and the outside is called the *master FSM*. A hierarchical FSM does not add expressiveness to the model of computation, as in fact it can be flattened. Consider the example in Figure 3.4(a). State A_1 is composed of two *substates* A_{11} and A_{12}. We assume that each time when a transition from A_2 to A_1 occurs, the system will always enter A_{11}, the initial state of A_1. Then we may derive the equivalent "flat" FSM as shown in Figure 3.4(b).



(a) (b)

Figure 3.4 Hierarchical FSM and its equivalent

A hierarchical FSM does not reduce the number of states, but it can significantly reduce the number of transitions and make the FSM more intuitive and easy to understand. However, a hierarchical FSM must be carefully constructed. The guard expression in the slave FSM and master FSM must be disjoint [8]. Otherwise, the system may have multiple transitions enabled, and it becomes non-deterministic.

3.3 Semantics of HDF

As we mentioned before, HDF is a heterogeneous composition of SDF with FSM. Ptolemy II uses *modal models* [15] to compose FSM with other models of computation hierarchically. A modal model is a composite actor with an FSM director as its local director. Figure 3.5 shows an example of an HDF model. Each state of the FSM is further refined to an SDF model. The FSM director connects and passes data between the current state refinement and the outside. Refinements in states other than the current state are disconnected from the system.

A *state of an HDF model* is identified by a different set of connections among dataflow graphs in hierarch. Each state of HDF corresponds to a different schedule for the system. The *initial state of an HDF model* is the state of HDF from which the model starts execution. For example, the HDF model in Figure 3.5 has two states, which can be identified by the states of the modal model actor B. For simplicity, we use $S = \{B_1, B_2\}$ to indicate the sets of states of this system. The initial state of the system is B_1.

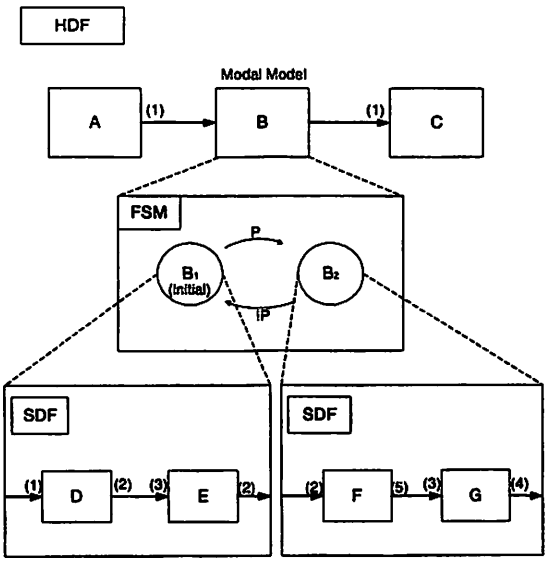


Figure 3.5 HDF model

The model in Figure 3.6 is a more complex example. An HDF model may consist of more than one modal model at the same level (such as A and B). Further, a state in the modal model (such as A_1) may also refine to another HDF model. The depth of hierarchy is arbitrary. In either of these two cases, the state of the system is determined jointly by the current states of these modal models. For example, the set of states of the HDF model shown in Figure 3.6 is:

$$S = \{ A_{11}B_1, A_{12}B_1, A_2B_1, A_{11}B_2, A_{12}B_2, A_2B_2 \}.$$

The initial state of the system is $A_{11}B_1$.

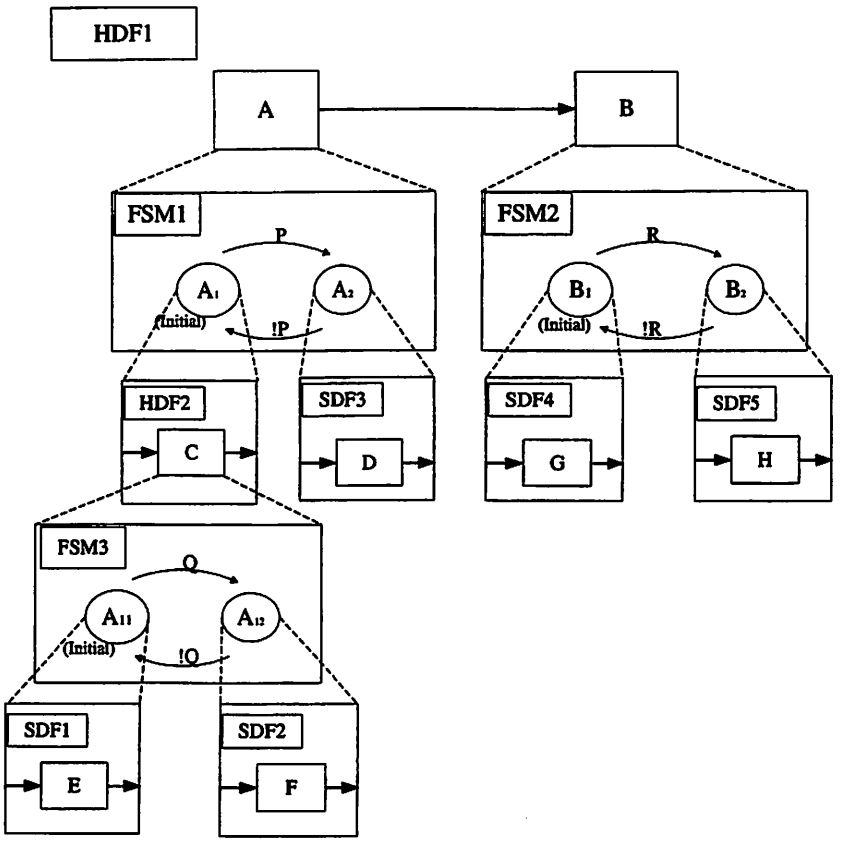


Figure 3.6 Actors *A*, *B* and *C* are modal models. *C* is embedded in *A*, while *A* and *B* are at the same level of hierarchy.

In HDF, a firing of the modal model is a *local iteration*, defined by the dataflow model in the current state refinement. In contrast, an iteration of the whole system is called a *global iteration*. The local iteration determines the rate signature of the modal model. Therefore, when a state transition occurs, the modal model may be configured to a different rate signature. This approach is intuitive. However, we must be careful about when a state transition may occur safely so that the graph remains consistent. First, the rate signature of the modal model actor should be well defined, which means it must remain constant during a complete firing. Furthermore, we would like to retain the

advantages of SDF, and the schedule should be able to remain valid during the execution. The simple case is where the rate signatures of all the states are same. If further, all the rate signatures are 1, the system is called a *homogenous* SDF system. Otherwise, it is called *non-homogenous* SDF system. In this case, we may allow a state transition to occur at the end of a firing of the modal model. The schedule for the higher hierarchy remains valid since the modal model appears as an opaque composite actor with constant rate signatures. There is no rescheduling issue and the existing SDF director is sufficient to handle this case.

The complication occurs if the rate signature of each state is different, as shown in Figure 3.5. A state transition corresponds to a new set of balance equations and a new definition of iteration. The semantics we choose for HDF is that each rate signature must remain constant during a global iteration, which is determined by the top level HDF director. This guarantees that a schedule for the whole system is always executed completely and actors always have sufficient data. The HDF model can be viewed as an SDF model within each state. When a state transition occurs, a new global schedule must be obtained.

The composition of HDF/SDF inside a modal model composite actor inside HDF may hide relations between actors and introduce deadlock, as we have discussed in Section 2.1. This is one disadvantage of HDF. But on the other hand, the exhibition of modes of operation in HDF has significantly increased the expressiveness of SDF.

3.4 Multi-Token Syntax

It is worth noting how we should develop a syntax for the guard expression in an FSM. The example of "$P > 0$" as we used in the previous section implies that there is at most one token on each firing of the modal model. This syntax is valid in CT, DE and many other domains composed with FSM. However, in SDF/HDF models, multiple tokens may be consumed and produced at each input and output port during a firing. A. Girault et al. proposed in [7] a syntax that borrows notation from the Signal language. For a port P, "P" denotes the most recent (last) token consumed/produced at port P, "$P\$1$" denotes the next most recent token, and "$P\$2$" denotes the next most recent. In the implementation of Ptolemy II, we use a slightly different syntax where "*P_array(i)*" is the $(i+1)$-th most recent token consumed/produced. For consistency with the general FSM in Ptolemy II, we also use "P" to denote the last (most recent) token, i.e., $P = P\_array(0)$. There is not much difference between the two notations. The choice is because we can easily borrow the existing syntax for array in Ptolemy II.

3.5 Implementation of HDF in Ptolemy II

We have implemented an HDF director and an HDFFSM director in Ptolemy II. The HDF director extends the SDF director to handle rescheduling when a state transition occurs. The HDFFSM director is an extension of the FSM director that works collaboratively with the HDF director. It implements the multi-token syntax and allows state transitions only between global iterations in HDF.

HDF Director

The computation of an HDF schedule is similar to the hierarchical scheduling in SDF opaque composite actors. In HDF, local SDF and HDF directors in the state refinement are responsible for computing their local schedules, which define the rate signatures of the modal model (composite) actors. The schedule computed by the top level HDF director is the global schedule.

Since HDF uses FSM to define different sets of rate signatures, it has only a finite number of schedules. Therefore, it is always possible to compute all the schedules prior to execution by finding all possible states of the HDF system. However, the number of such possible schedules grows exponentially with the number of modal models [7]. Therefore, this approach is only suitable for HDF with a small number of states.

The alternative, which is what we have implemented in Ptolemy II, is to compute the schedules dynamically when a state transition occurs. Once a new schedule is obtained, the HDF director will cache the schedule labeled by its rate signatures. The cache is a queue that places schedules in the ascending order with the most recently used at the end of the queue. When a state is revisited, we could use the schedule identified by its rate signatures in the cache, and move it to the end of the cache. Moreover, if two states have the same rate signatures, then only one schedule is necessary. (Consider the case when SDF is composed with FSM. No rescheduling is needed.) The size of the cache can be set

as a parameter in the HDF director based on the availability of memory. If the cache is full, the least recently used schedule (placed at the beginning of the queue) is discarded.

Computing the schedule dynamically rather than prior to execution can avoid unnecessary computation for states that do not occur in execution. This often occurs in communication and distributed systems, where components are scattered in space. Each component implements modes of operation using FSM, but the whole system is only allowed to work in some *collaborative modes*. For example, in a single two-way channel communication between actor *A* and *B*, each actor implements a *send* mode and a *receive* mode. But the system will only work correctly when either *A* sends message and *B* receives, or *B* sends and *A* receives. The other two states when both actors send or receive should never occur. This can be controlled by sending some control signal to both actors, which is not known to the system prior to execution.

HDFFSM Director

In the HDF domain, the HDFFSM director handles state transitions by calling its requestChange() method. This method is used to queue topology changes (called *mutations*) with the *manager* in Ptolemy II. A manager [1] controls the overall execution of a model and executes mutations when changes can safely occur. In the implementation of manager in Ptolemy II, mutations are executed between global iterations. This mechanism is used in HDF to guarantee that state transitions are chosen only after an iteration finishes.

The HDFFSM director implements the multi-token syntax by setting up a buffer for each port of the modal model. The buffers cache all the tokens consumed/produced in one iteration. The multi-token syntax can also be used in non-homogeneous SDF composed with FSM. In this latter case, the HDFFSM director allows state transition between arbitrary firings of the modal model. The buffers will only cache all the tokens in each firing.

4. HDF Examples and Discussions

4.1 Communication Examples

Switch coding in a wireless channel

We now go back to the example where an encoder can switch between two coding algorithms. For simplicity, we use a (7, 4) Hamming code and a (3, 1) Hamming code (also called a repetition code). The (3, 1) code has higher processing gain but lower efficiency. Further, we assume a Boolean signal is available to the encoder to indicate the switching. If the signal is "true," the encoder will use the (3, 1) Hamming code; otherwise it will use the (7, 4) Hamming code. Such a signal might come from a sensor, a performance detector, or a command sent by a person.

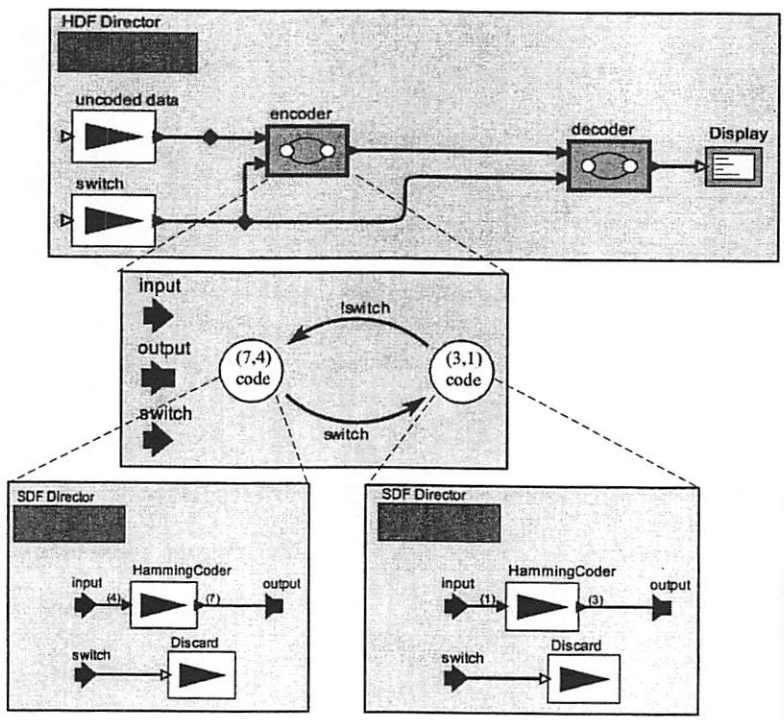


Figure 4.1 An encoder that can be switched between two coding schemes. The corresponding decoder has a similar architecture and is not shown here.

If further, we send the switching signal to the decoder side, we may construct a decoder that switches between the two decoding schemes. The switching is synchronous with the encoder. The implementation in HDF is shown in Figure 4.1. The Discard actors in the state refinements are used to define the consumption rates of the *switch* port to be 1. In SDF, if a port of the composite actor is not connected to anything from inside, its port rate is considered to be zero.

Data-dependent resampling

The Ethereal Sting example in Figure 3.2 turns out to have a different solution from the switch coding example. Figure 4.2 shows the resampling diagram in a simpler form. Based on the system analysis, the rest part of the system can be viewed as an SDF actor with a production rate of 32768 on both the base band signal and the control signal. This means that if we use HDF to implement flow control, the modal model cannot make state transition until there are 32768 tokens consumed. This is certainly not what we want.

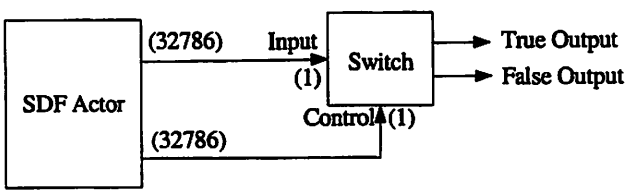


Figure 4.2 A simpler form of the data-dependent resampling.

Fortunately, the consumption rate of the BooleanSwitch *input* is always 1, independent of the control token value. Therefore, we may use SDF composed with FSM, as shown in Figure 4.3. The modal model consumes one input token and one control token on each of

its firing. Although it will fire 32768 firings per iteration, state transition is allowed to occur between arbitrary firings. If the *control* is "false," the modal model is in the *falseState*, and the input token is discarded. If the *control* is "true," the modal model is in the *trueState*, and the input token is processed to send to the plotter. A unit delay is introduced in the input token sequence. This is because state transition (and hence the control token) must occur before the input token arrives.

This method can be generalized if the consumption rate of the BooleanSwitch *input* is constant, and if the graph remains consistent. If the consumption rate is dependent on the value of the *control*, we cannot use this method.

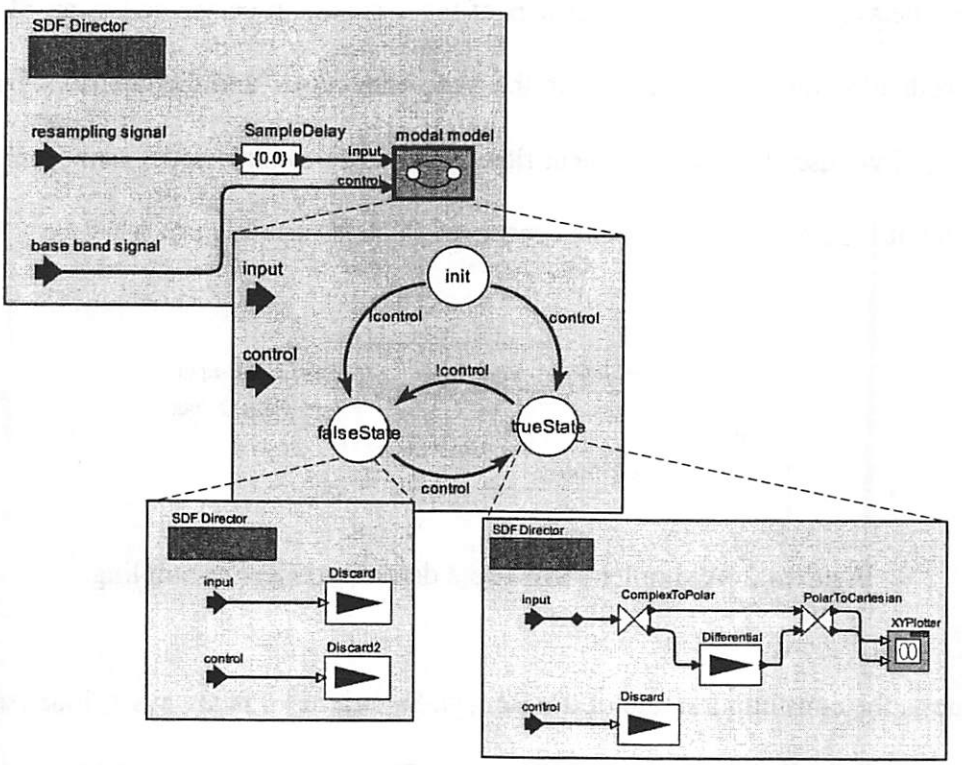


Figure 4.3 Data-dependent resampling in the Ethereal Sting project

4.2 Extension: HDF with Infinite Rate Signatures

The original definition of HDF proposed by A. Girault et al. in [7] states that HDF has a finite number of rate signatures, since each state of the finite state machine refines one rate signature. However, consider the encoder part of the switch coding example in Section 4.1. The two states use the same architecture and same "HammingCoder" actor in the Ptolemy library. The only difference is their parameter configurations. Therefore, we may think of using only one state with a "HammingCoder" actor in the refinement, and associating a reconfiguration *action* with the state transition. Each time a transition occurs, the state is reconfigured to a *new* state.

This suggests that in theory it is possible that an HDF model may have an infinite number of rate signatures with only finite number of states and transitions. However, since rate signatures can be set to arbitrary values during execution, this extension of HDF will make consistency and deadlock undecidable very likely.

We use the following example to show that the number of states and transitions can be far smaller than the number of rate signatures. Consider the model in Figure 4.4, which mainly has one state and a self-loop transition. The initial state here only serves as a starting point to give the initial parameters. The downsampling factor in the n-th iteration is set to be the output of the $(n-1)$-th iteration. When a sequence of natural numbers is sampled, the sampling rate is doubled each time a transition occurs. This will generate the sequence of powers of 2.

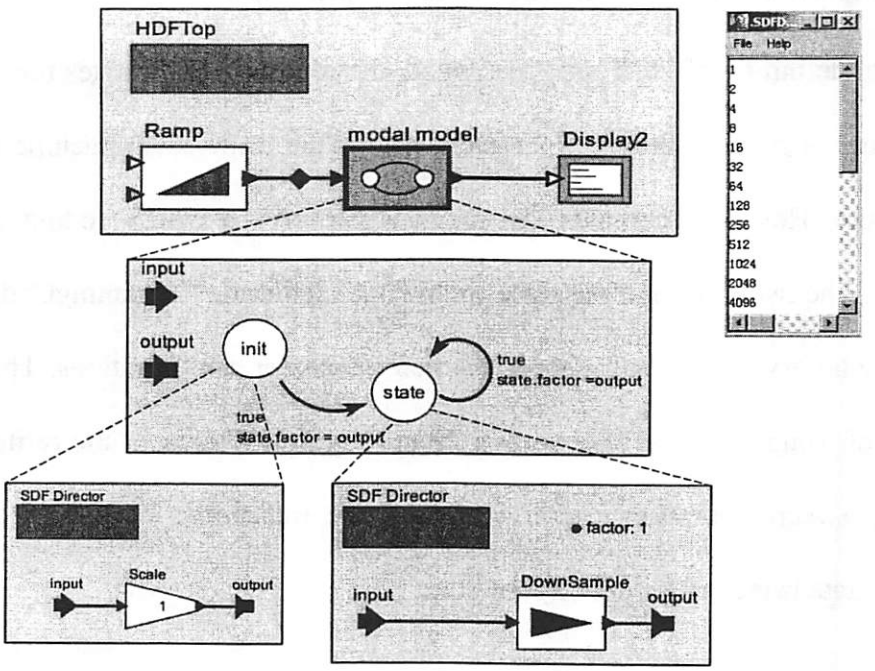


Figure 4.4 HDF with infinite rate signatures

Given another input sequence, the sampling rate and output in each iteration may be totally different. For this kind of system, rate signatures are dependant on data inputs and outputs of the system. It is therefore impossible to compute all the possible schedules at compile time. This shows another advantage of computing the schedule dynamically in HDF.

4.3 HDF and Parameterized SDF (PSDF)

The reconfiguration process of the two examples in Section 4.2 is similar to a parameterized SDF (PSDF) model, where the values of parameters can be changed at run time. A PSDF model Φ consists of three graphs – the *init graph* Φ_i, the *subinit graph*

Φ_s, and the *body graph* Φ_b [2] [3]. The body graph describes the main dataflow model. The init graph and subinit graph configure the parameters of the body graph. Therefore, PSDF allows more complicated reconfiguration process by using two graphs of Φ_i and Φ_s, rather than a simple action of state transition in HDF.

The motivation of providing two graphs Φ_i and Φ_s for configuration is to distinguish between two kinds of parameter reconfigurations when PSDF is embedded as a composite actor in a hierarchy [3]. Parameter reconfigurations that may affect the dataflow of the higher hierarchy are performed in the init graph Φ_i, which is executed once at the beginning of a global iteration. Parameter reconfigurations that do not affect dataflow of the higher hierarchy can be performed in the subinit graph Φ_s, which is executed at the beginning of each execution of Φ. The body graph Φ_b is executed after each firing of Φ_s. B. Bhattacharya and S. S. Bhattacharyya use a downsampler example in [3] to illustrate this. The *factor* parameter of the downsampler, which affects the consumption rate, is reconfigured in Φ_i. The *phase* parameter, which is allowed to happen more frequently, is reconfigured in Φ_s.

The different usage of Φ_i vs. Φ_s is analogous to HDF vs. SDF composed with FSM. In HDF, state transitions can only occur between global iteration because the rate signature of each state is different from others. In SDF composed with FSM, state transition can occur between arbitrary firings of the modal model, since rate signature of the composite

actor remains constant. This similarity between PSDF and HDF is not surprising, since the semantics of both models of computation are designed in the way to ensure consistency in dataflow graphs when change occurs. As a comparison, PSDF is more expressive in parameter change. It is hard to use HDF to simulate *both* the Φ_i and Φ_s processes of PSDF, even for the simplest model such as the downsampler example in [3]. But HDF is more expressive in change of model structure. For example, the switching coding example using HDF in Section 4.1 does not need to implement two Hamming codecs. It could use a Hamming code in one mode, and a convolutional code in the other.

4.4 HDF and CSDF

As we mentioned in Section 2, a CSDF [4] actor varies its rate signature between firings in a finite cycle. In HDF, rate signature changes do not have to appear cyclically, nor does the number of different rate signatures have to be finite. The firing rules for CSDF and HDF are also different. A CSDF actor changes its rate signature after each of its firings, which is independent of other actors it is connected to. In HDF, a modal model actor can only change rate signatures between global iterations. Rate signature changes are more restricted in HDF than in CSDF. The model structure of HDF is also more complex than CSDF. If a system works in N modes in a cyclic manner, then it is more efficient to use CSDF than HDF. HDF is more expressive when rate signature changes are data-dependent.

5. Conclusions and Future Work

SDF is useful in design and modeling of multi-rate signal processing algorithms. An SDF schedule can be computed at compile time, and questions such as deadlock and bounded memory remain decidable. Many communications and signal processing actors are implemented in Ptolemy II based on the SDF semantics. But SDF does not allow rate changes during execution.

HDF is a heterogeneous composition of SDF and FSM. Actors may change rates in HDF by state transitions in an FSM. State transitions may only occur at the end of one global iteration. This is a key constraint that ensures many properties of SDF are retained in HDF. We have implemented an HDF director and an HDFFSM director based on the existing SDF and FSM infrastructures in Ptolemy II. The multi-token syntax in the HDFFSM director can also be used to construct non-homogeneous SDF composed with FSM.

HDF provides a model of computation for signal processing systems with flow control. All the SDF actors can be used and reconfigured in HDF. We showed several examples using HDF and compared HDF with CSDF and PSDF. Future interesting work may include:

- Explore HDF on more complicated communications and signal processing systems.

- Extend to a timed-version of HDF.
- Study in more depth the comparison of HDF and other dataflow models such as BDF and DDF.
- Study the composition of HDF with other domains, such as DE and CT.

6. References

[1] Shuvra S. Bhattacharyya, Elaine Cheong, John Davis II, Mudit Goel, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi , Steve Neuendorffer, John Reekie, Neil Smyth, Jeff Tsay, Brian Vogel, Winthrop Williams, Yuhong Xiong, Yang Zhao, Haiyang Zheng, "Heterogeneous Concurrent Modeling and Design in Java," Vol 1 - 3, Technical Memorandum UCB/ERL M03/27 - M03/29, University of California, Berkeley, CA USA 94720, July 16, 2003.

[2] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized Dataflow Modeling of DSP Systems," Proc. of ICASSP, Istanbul, Turkey, June 2000.

[3] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized Dataflow Modeling for DSP Systems," IEEE Transactions on Signal Processing, vol. 49, No. 10, October 2001.

[4] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-Static Data Flow," IEEE Transactions on Signal Processing, vol. 44, February 1996.

[5] J. T. Buck and E. A. Lee, "Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model," Proc. of ICASSP '93, Minneapolis, MN, April 1993.

[6] J. T. Buck and E. A. Lee, "The Token Flow Model," presented at Data Flow Workshop, Hamilton Island, Australia, May 1992.

[7] A. Girault, B. Lee, and E. A. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models," IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems, vol. 18, No. 6, June 1999.

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

[9] B. Lee and E. A. Lee, "Hierarchical Concurrent Finite State Machines in Ptolemy," Proceeding of International Conference on Application of Concurrency to System Design, pp. 34-40, Fukushimia, Japan, March 1998

[10] B. Lee and E. A. Lee, "Interaction of Finite State Machines with Concurrency Models," Proceeding of Thirty Second Annual Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, California, November 1998.

[11] E. A. Lee, "Overview of the Ptolemy Project," Technical Memorandum UCB/ERL M03/25, University of California, Berkeley, CA, 94720, USA, July 2, 2003.

[12] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," IEEE Transactions on Computers, vol. C-36, No. 1, Jan. 1987.

[13] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," Proceedings. of the IEEE, vol. 75, No.9, pp. 1235-1245, September, 1987.

[14] E. A. Lee and D. G. Messerschmitt, "Digital Communication," Second Edition, Kluwer, 1994.

[15] X. Liu, J. Liu, J. Eker and E. A. Lee, "Heterogeneous Modeling and Design of Control Systems," Software-Enabled Control: Information Technology for Dynamical Systems, Tariq Samad and Gary Balas (eds.), Wiley-IEEE Press, April 2003.

[16] J. G. Proakis, "Digital Communications," Fourth Edition, McGrall-Hill, 2001.

[17] T. M. Parks, J. L. Pino and E. A. Lee, "A Comparison of Synchronous and Cyclo-Static Dataflow," Proc. IEEE Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA October 29 - November 1, 1995.

[18] The Ethereal Sting website: http://www.etherealsting.org/