Static Dataflow and Heterochronous Dataflow with Hierarchical FSMs in Ptolemy II

Brian K. Vogel
vogel@eecs.berkeley.edu
EE249 Final Project Report

December 9, 1999

Abstract

*charts is a formalism for combining finite state machines (FSMs) with various concurrent models of computation. In *charts, the emphasis is on separating the FSM semantics from the concurrency model semantics. Instead of defining concurrency models, *charts shows how to combine hierarchical FSMs with various existing concurrency models. This paper discusses an implementation of a subset of the *charts formalism under Ptolemy II. The particular subset implemented consists of static dataflow (SDF) combined with hierarchical FSMs and heterochronous dataflow (HDF) combined with hierarchical FSMs.

1 Introduction

1.1 Overview of Ptolemy II

Ptolemy II [1] is software (implemented in Java) supporting heterogeneous, concurrent modeling and design for embedded systems. Ptolemy II uses a component view of design, where a model is constructed as a set of interacting components. A model of computation is then used to describe the semantics of the interaction between the components. In Ptolemy II, a model of computation is implemented by a domain. Ptolemy II currently contains domains for several models of computation, such as static dataflow (SDF) and discrete events (DE). One of the objectives of Ptolemy II is to support the construction of heterogeneous models, i.e., models that use more than one model of computation.

1.2 Project Goals

Prior to this project, some Ptolemy II domains, such as SDF and DE, could be combined. However, other arguably useful combinations, such as SDF and FSM could not. At the onset of this project, the FSM domain itself was in an early state of development by another student, Xiaojun Liu. The SDF domain, however, was considered to be fairly mature. The primary objective of this project was to implement an interaction semantics between SDF and FSM, allowing Ptolemy II models to combine these models of computation. Another objective was to implement an interaction semantics between heterochronous dataflow (HDF) [2] and FSM. HDF, a generalization of SDF and cyclostatic dataflow (CSDF), is a somewhat experimental model of computation. The *charts

[2] formalism was used to describe the semantics of the interaction between SDF (or HDF) and hierarchical FSMs. A final objective was to construct some (hopefully interesting) demos and test cases using SDF/HDF with FSMs to verify the correct functioning of the code.

2 Motivations for Combining FSMs with Concurrency Models

Finite state machines are excellent for describing control flow. However, many complex systems require numerical computations in addition to control flow. Concurrency models such as dataflow are typically better suited to performing numerical computations. For many practical systems, it is therefore necessary to combine FSMs with concurrent models of computation.

There are a number of models that combine FSMs with concurrent models of computation. Examples include Harel's Statecharts model [3], codesign finite state machines (CFSMs) [4], The MathWork's Simulink, Hybrid systems [5], and *charts (pronounced "starcharts") [2]. Unlike the other models mentioned above, *charts does not define a concurrency model. Rather, *charts defines an interaction semantics for FSMs and several existing concurrency models. This is exactly what is desired in Ptolemy II, since it would allow Ptolemy II models to combine the FSM domain with one or more concurrent domains. That is, in Ptolemy II, one might wish to construct a heterogeneous model that composes FSMs with another concurrency model such as SDF. It would be desirable to be able to describe a model using a single FSM domain and the concurrent domain(s) of choice instead of requiring the use of a separate specialized domain for each combination of FSM and concurrency model.

This project involved implementing the *charts interaction semantics for hierarchical FSMs with SDF and hierarchical FSMs with HDF. Another student, Xiaojun Liu, is currently implementing the *charts interaction semantics for the other Ptolemy II concurrent domains (Discrete Events and Continuous Time).

3 Semantics of SDF with Hierarchical FSMs

In *charts, there are two possible ways to combine SDF with FSMs. In the first case, an actor in an SDF graph may refine to an FSM. In the second case, a state in an FSM may refine to an SDF graph. In *charts, an FSM must externally have the semantics of its master. Therefore, if SDF is to be combined with hierarchical FSMs, then the top level of a model must be SDF.

Apart from the above constraint that the top level be SDF, *charts supports arbitrary nesting of SDF and FSM subsystems in a model. Figure 3 shows an example of SDF combined with FSMs.

3.1 SDF Refining to FSM

*charts specifies an operational semantics for the case where an SDF actor in an SDF graph refines to an FSM. When an SDF actor refines to an FSM, the FSM must externally have SDF semantics. The slave FSM must therefore consume a (fixed) number of tokens from each port, corresponding to the consumption/production rates of its master, when is fired.

When execution begins, the FSM will start in an initial state. State transitions are allowed to occur only between iterations of the SDF graph in which the FSM is embedded. The semantics is that when an FSM is fired, it fires its current state (which may refine to another FSM or an SDF diagram). The FSM will consume and produce tokens on its input and output ports,

respectively, corresponding to the type signature of its master. After the last firing of the iteration, the guard expressions of the current state's outgoing transitions are evaluated. If a transition's guard expression evaluates to true, a state transition will then occur.

3.2 FSM Refining to SDF

*charts specifies an operational semantics for the case where a state in an FSM digram refines to an SDF subdiagram. Since the FSM must externally obey SDF semantics, it is required that an FSM state's refinement have a the same type signature as the SDF actor that the FSM refines. For example, in Figure 3, both of the states refine to SDF actors with the same type signature as the FSM's master. Note that the "Const" actors implicitly contain an input port with a consumption rate of one, even though it is not explicitly shown in the figure.

4 HDF with Hierarchical FSMs

Consider the case where an SDF actor refines to an FSM and each of the FSM states refines to SDF diagrams. It is required that all of the refining FSM states have the same type signature (that of the top level SDF actor). An interesting generalization occurs when the refining states are allowed to have distinct type signatures. Since the FSM externally has the type signature of its current state's refinement, this means that the FSM no longer obeys SDF semantics. The FSM now has a finite number of type signatures corresponding to the distinct type signatures of its refining states. The external semantics of such an FSM is that of the heterochronous dataflow (HDF) model of computation [2]. HDF is a generalization of SDF since the type signature of an actor is allowed to change. HDF is similar to cyclo-static dataflow (CSDF) in that an actor has a finite number of distinct type signatures. Unlike CSDF, however, an HDF actor is not restricted to cycling through its type signatures. HDF is therefore also a generalization of CSDF.

In HDF, an actor is only allowed to change its type signature between iterations. In general, the order in which the different type signatures are used is not predictable. In the *charts formalism for hierarchical FSMs with HDF, the type signature may change when a state transition occurs in the refining FSM. Note that it is not necessarily the case that an HDF actor must explicitly refine to an FSM. For example, a language with imperative semantics, such as Java, may be used to implement an HDF actor.

The HDF model of computation is interesting because it is more general (and more expressive) than SDF, yet HDF retains the desirable decidability properties of SDF. Since each actor in an HDF diagram has only a finite number of type signatures which can only change between iterations, it is possible to compute all possible schedules statically (at compile-time). As a result, deadlock and bounded memory are decidable. Note that the number of schedules may increase exponentially with the number of actors.

One possible drawback to HDF is that it may be difficult (non-intuitive) to use, since a global schedule determines when state transitions are allowed to occur. Another potential drawback is that there might not be many interesting applications for which HDF can represent the model elegantly. It remains to be determined whether there exists a sufficiently large class of systems for which there are compelling reasons to use HDF with FSMs over SDF with FSMs.

Figure 1 shows one interesting system implemented using HDF. Given two strictly increasing sequences of integers, the (deterministic) system combines the two sequences to produces a single

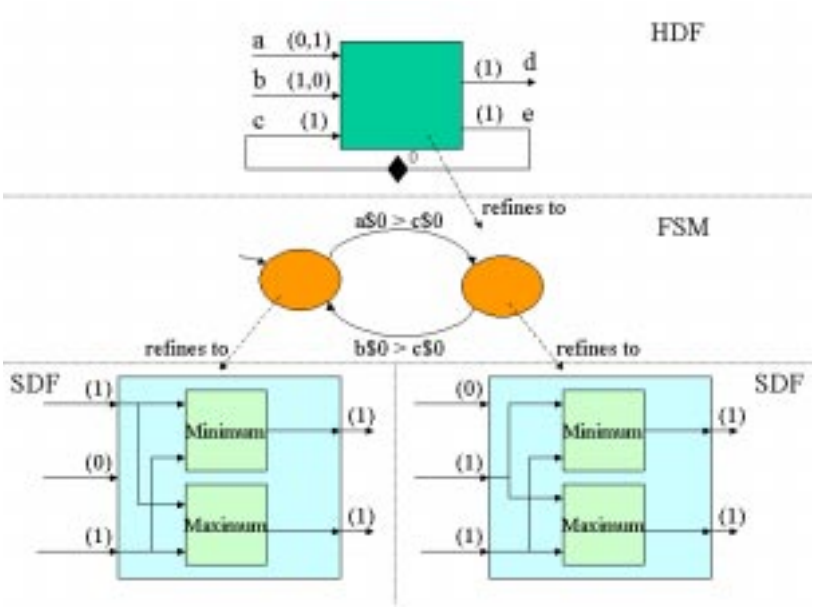

Figure 1: HDF system that performs a (deterministic) merge of two strictly increasing integer sequences into a single strictly increasing sequence. The numbers in parenthesis denote the tokens consumption and production rates. Note that the top level HDF actor has two distinct type signatures.

strictly increasing sequence on its output. Figure 2 shows a trace of this system for one possible pair of input sequences. This system illustrates that one potentially useful feature of HDF is the ability of an actor to have ports that alternate between zero and non-zero consumption or production rates. This example illustrates the increase in expressiveness that HDF offers over FSMs combined with SDF. The system in Figure 1 can be implemented in an elegant way using hierarchical FSMs with HDF. However, it is not possible to represent this system using *charts hierarchical FSMs with SDF. This system can be represented using the more general dynamic dataflow or process networks models of computation. However, these more general models of computation lack the desirable decidability properties of HDF.

5 Ptolemy II Implementation

For this project, a subset of the *charts formalism was implemented under Ptolemy II. The particular subset implemented consists of static dataflow (SDF) combined with hierarchical FSMs and heterochronous dataflow (HDF) combined with hierarchical FSMs.

5.1 State of Ptolemy II at Project Onset

At the start of this project, Ptolemy II had a fairly mature SDF domain (implemented by Steve Neuendorfer) and partially functioning FSM domain (implemented by Xiaojun Liu) in an early state of development. These domains could be used separately, but not combined to create models.

d	0	1	2	3	5	20	29	30	50	100	101	109
a	1	x	2	3	5	100	x	x	x	x	109	x
b	x	20	x	x	x	x	29	30	50	101	x	200
c	0	1	20	20	20	20	100	10	100	100	101	109
state	s0	s1	s0	s0	s0	s0	s1	s1	s1	s1	s0	s1
e	1	20	20	20	20	100	100	100	100	101	109	200

Figure 2: One possible trace for the HDF model in Figure 1. The pair of input sequences are: a={1, 2, 3, 5, 100, 109}, and b={20, 29, 30, 50, 101, 200}. The output sequence is produced on d.

5.2 Implementation Details

Most of the implementation work of this project consisted of extending and enhancing the FSM domain to implement *charts interaction semantics when FSMs are combined with SDF. It was then relatively straightforward to allow FSMs to be combined with HDF.

The current implementation allows arbitrary nesting of FSM and SDF/HDF subdiagrams. The main constraints are that the top level of a model must be an SDF or HDF diagram, and that all FSM states must eventually refine to an SDF or HDF subdiagram. This behavior is required since *charts requires that an FSM must externally obey SDF or HDF semantics when FSM is combined with SDF or HDF, respectively.

This implementation uses the Ptolemy II expression language to express the guard expressions of state transitions. For this project, the FSM domain was extended to allow a token syntax in guard expressions similar to that described in [2]. In the current implementation, it is possible for transition guard expressions to reference a history of tokens read or written through an FSM's containing actor. A port name is followed by a "$" and a nonnegative integer to specify its position in the history. As an example, suppose that an SDF actor with an input port called "a" refines to an FSM with the guard expression: "a$0 && a$2". In this example, "a$0" refers to the token most recently read in port "a" and "a$2" refers to the third most recently read token. If "a" is a boolean input, then a state transition will occur if the guard expression evaluates to "true" after the final firing of an iteration.

Several simple test case demos have been run to verify the correct functioning of the implementation.

5.3 Demos

A somewhat interesting Ptolemy II demo of hysteresis that uses hierarchical FSMs with SDF is shown in Figure 3. Figure 4 shows a plot of the input signal (blue) and the output signal (red).

6 Conclusions

In this project, the *charts interaction semantics for hierarchical FSMs with SDF, and hierarchical FSMs with HDF were implemented in Ptolemy II. Several test cases have been constructed to test the correct functioning of the implementation, and an interesting (but simple) demo applet has been created.

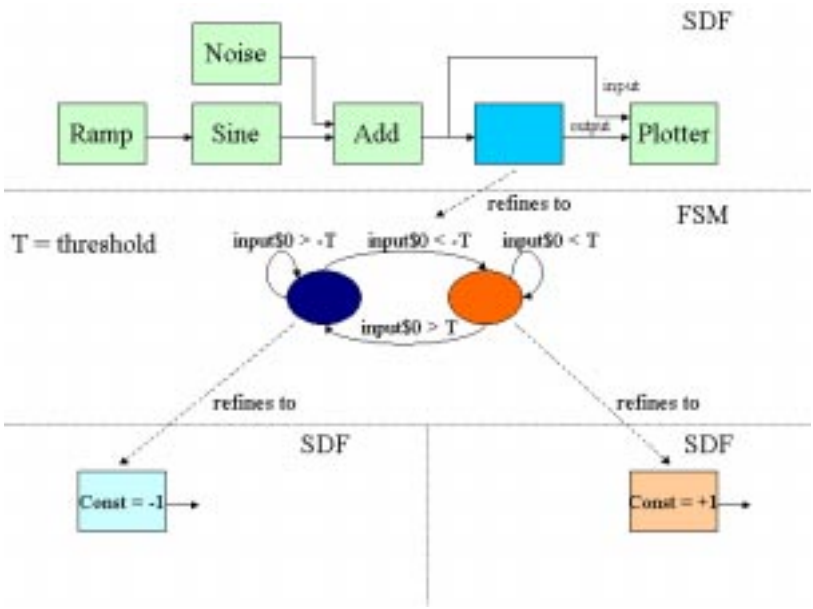

Figure 3: Hysteresis system using hierarchical FSMs with SDF. Note that all SDF actors are homogeneous.

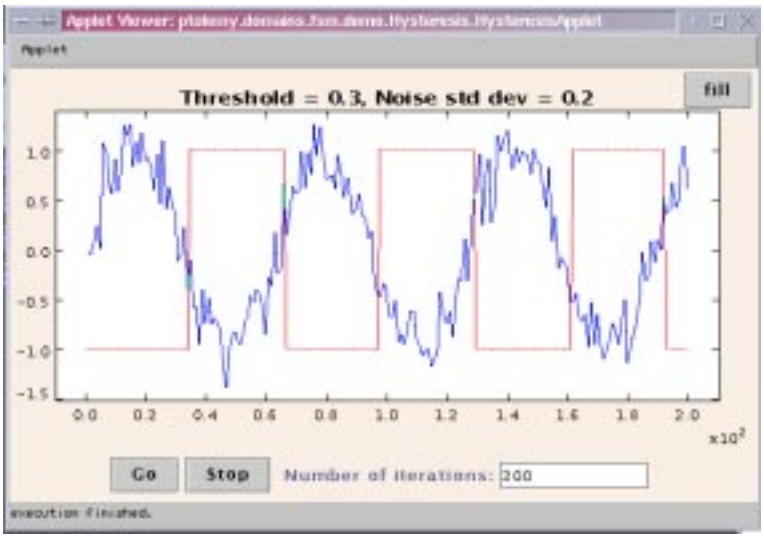

Figure 4: Plot of the output of the system in Figure 3. The input signal is shown in blue, and the output signal is shown in red.

There are several possibilities for future work. One area is performance optimizations. In some cases, token transfer can involve a large overhead. For the HDF case, it may be interesting to investigate alternative scheduling methods, such as computing all possible schedules at compile-time, or caching schedules computed dynamically. It my also be desirable to precompute all possible schedules for the purpose of determining whether deadlock can occur or whether bounded memory usage is possible. The current implementation computes schedules dynamically, but does not cache them.

For the HDF case, it may be useful to allow a port of an HDF actor to alternate between zero and non-zero consumption/production rates, as is done in the system in Figure 1. The current implementation does not allow a port to have a consumption or production rate of zero. Only positive integer rates are currently allowed.

For both the case of hierarchical FSMs with SDF and hierarchical FSMs with HDF, it may be interesting to consider changes to the state transition guard expression syntax and semantics. It is hoped that the process of implementing several interesting and/or practical designs using this implementation will lead to insights that will lead to improvements in the implementation. Although several test cases have been constructed to test more sophisticated implementation features such as non-homogeneous SDF and HDF actors, and several levels of hierarchy, interesting demo applications that use these features have not yet been created.

References

[1] 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 and Yuhong Xiong, "Overview of the Ptolemy Project," ERL Technical Report UCB/ERL No. M99/37, Dept. EECS, University of California, Berkeley, CA 94720, July 1999.

[2] A. Girault, B. Lee, and E. A. Lee, "Hierarchical Finite State Machines with Multiple Concurrency Models," April 13, 1998.

[3] D. Harel, Statecharts: A visual formalism for complex systems, Sci. Comput. Program., vol. 8, pp. 231 274, 1987.

[4] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli, Hardware-software codesign of embedded systems, IEEE Micro, pp. 26 36, Aug. 1994.

[5] T. A. Henzinger, The theory of hybrid automata, in Proc. 11th Annu. IEEE Symp. Logic in Computer Science (LICS), 1996, pp. 278 292.