Design of Fault Tolerant Data Flow in Ptolemy II

Mark L. McKelvin, Jr.
mckelvin@eecs.berkeley.edu

EECS 290N Report
December 17, 2004

University of California at Berkeley
Berkeley, CA 94720, USA

Abstract

Designing embedded software for complex, safety critical, real-time feedback control applications is a complex task. Typical applications, like a steer-by-wire application, contain a model of the components computing control laws and interacting with a plant using sensors and actuators. Well-defined mathematical models are often useful in the design of such systems because they allow formal validation, techniques like code generation, and reduces the ambiguity in specifications amongst a team of designers. An experimental model of computation called Fault Tolerant Data Flow is explored for safety critical, real-time feedback control systems. This report describes the operational semantics and structure of this model of computation, and its implementation in the Ptolemy II design environment.

1 Introduction

Designing software for complex, safety-critical feedback control systems has prompted researchers and scientists to look at fault tolerant design methodologies to describe the behavior of such systems under fault coverage and design cost constraints [1]. In the automotive domain one example where failure to a sub-system, like a steer-by-wire subsystem, may yield disastrous results. In [1], an automatic synthesis-based approach is taken to design a methodology that allows a designer to explore the trade-off between cost and fault-coverage, and it is based on a formal mathematical model, or model of computation, that is amenable for specifying fault-tolerant, periodic feedback control systems.

Fault-tolerance is often viewed as an approach to increasing the reliability property of safety-critical systems. The approach assumes that components will inevitably lead to faults or system failure, but it attempts to neutralize the inevitability of faults that leads to system failures by employing redundancy in system components. Furthermore, executions of applications in this domain are expected to execute periodically on a possibly infinite stream of input data on an execution platform within bounded memory. An execution platform is a hardware/software system that may include software components, such as a real-time operating system or middleware services, and a hardware layer that may include a processor, drivers, and communication channels.

Fault Tolerant Data Flow (FTDF) is an experimental model of computation designed to address the specification of fault-tolerance in safety-critical, real-time feedback control systems, and it is first introduced in [1]. FTDF is a data flow variant that targets efficient executions and alleviating ambiguous specifications in the modeling, design, and validation phases of a fault-tolerant system. The structure of FTDF enables formal analysis and automatic synthesis tools and techniques. In this paper, FTDF structure and operational semantics are reviewed. Finally, an implementation of the FTDF domain in the Ptolemy II design environment for modeling, simulation, and code generation is discussed.

The rest of this paper is organized below. Section 2 discusses previous work with FTDF and a closely related model of computation, Statically Scheduable Data Flow. Section 3 reviews the operational semantics and structure of an FTDF graph. Section 4 discusses the implementation of FTDF in Ptolemy II and gives a simple example. Section 5 concludes with concluding observations and future work.

2 Previous Work

In [1], FTDF is the founding programming model of an automatic synthesis-based methodology for the design of real-time, safety-critical feedback control applications. The purpose of the programming model is to provide a model of computation for which design environments can be built that formally analyzes and validates safety-critical applications. A library for the specification of FTDF applications is developed for use in the Metropolis Meta Model design environment [4].

FTDF is closely related to Statically Schedulable Data Flow (SSDF), also known as Synchronous Data Flow [2], [3]. SSDF has been used to describe signal-processing systems. Under SSDF, functional units, or *actors* [6], are scheduled statically prior to run-time. That allows for a more efficient execution of a possibly infinite sequence of inputs that is guaranteed to run in bounded memory and will execute without deadlocks. FTDF is designed to allow bounded memory and deadlock-free executions, but it also allows the designer to design actors that can accept a subset of inputs. This violates the ability to use balance equations to determine a completely static schedule before run-time, as is done in SSDF. The work presented in this paper will leverage an existing set of expressive components in the Ptolemy II design environment [5] to allow users to experiment with the use of a FTDF domain. The goal is to allow the specification and simulation of an FTDF model that may be integrated with other validation tools or used for code generation.

3 Operational Semantics

This section reviews the fundamental components of a FTDF model. The fundamental components include tokens, actors, and communication media. These components are discussed in terms of their operational semantics in a precisely defined FTDF graph.

3.1 Tokens

Tokens are encapsulations of data. This allows for type polymorphism on the data that is passed between actors. In addition, in the FTDF domain, tokens are appended with a *valid* field. This token field is used as the result of an actor that may perform an error detection scheme, or a designer who wishes to set the valid field based on whether or not an invalid token is produced by an actor. An invalid token may be a token with an invalid value due to some error or fault detected at an actor. In this case, an actor receiving a token can check the token's validity and choose whether or not to use the token.

3.2 Actors

In general, actors are functional components that consume a finite number of input tokens per firing and produces a finite number of output tokens over a possibly infinite sequence of firings. Formally, we can define an actor, as given in the lecture notes, as being a data flow process where $F: S^n \to S^m$, as a mapping between a n-tuple set of input signals to a m-tuple set of output signals where $S = T^{**}$ and T^{**} is the set of finite and infinite sequences, including $\perp_n$, of data type T. In addition a firing rule that dictates when an actor can fire, given as $U \subset S^n$ such that $\forall u \in U$, each u is finite and no two elements of U are joinable, and a firing function, $f: S^n \to S^m$, such that $\forall u$, $f(u)$ is defined and finite. These elements yield a data flow process, F such that $F(s) = f(u).F(s')$ if $\forall u \in U$, such that $s = u.s'$, otherwise $F(s) = \perp_n$, where $\perp_n \in S^n$ is the n-tuple of empty sequences. Given that definition of data flow process, an actor can be defined by a firing function that fires on a valid set of firing rules. Repeatedly firing an actor to find a least point based on the actor's firing function such that the firing rules are satisfied, precisely defines the operational semantics of a data flow process. For FTDF, a repeated firing of all actors in the model defines the operation of the FTDF model.

Actors are typed, and as such, only four types of actors will be briefly discussed in this paper. Two types of actors result from actor-oriented languages. These are source and sink actors. They represent sensors and actuators respectively. Regular actors have inputs and outputs, and the firing rule for regular actors typically must fire when all inputs are available. More formally, the typical firing rule for regular actors is $u_1 = [*, *, \ldots, *]$ for an N-input regular actor where $u_1 \in U$. Firing on all inputs is typical of other data flow languages. The final type of actor is the *input* actor. The input actor can fire on a subset of inputs. An example of an input actor that may fire if at least two of three inputs are available is shown in Figure 1. The set of possible firing rules are, $U = [\{*, *, *\}, \{\perp, *, *\}, \{*, \perp, *\}, \{*, *, \perp\}]$.

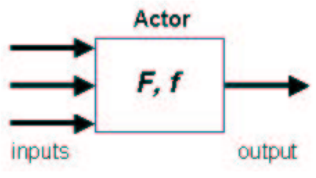


Figure 1: This is an example of an *input* actor with three inputs that may fire if at least two inputs are available.

3.2 Communication Media

Communication media acts as unidirectional channels that transmit tokens between actors. Communication media is the same as arcs in SSDF graphs, and as such, are implemented in Ptolemy II with bounded buffers. In real applications, communication media are potential sources of error. For simplification in modeling in the Ptolemy II environment, communication media are assumed to transmit tokens from source to receiver with no faults in the media.

3.3 Rules of Composition

The composition of actors and communication media can be formulated as a directed, acyclic graph called a FTDF graph. This defines the semantics of the composition of actors and communication media. These semantics can be viewed as a set of constraints on the model that must be obeyed by the designer to yield an implementation of FTDF. Given a set of actors, say A, and a set of communication media M, a FTDF graph, G is given as $G = (V, E)$, where V is the set of vertices where $V = A$ and E, the set of directed edges connecting V, is $E = M$. Furthermore, it is important to note that in the FTDF semantics, a FTDF graph is *legal* if the following conditions hold:

- Graph G contains no causality cycles
- $\forall$ v $\in A_I$, where A_I = actors of type *input*, inputs can come from actors of the source or regular actors
- A cycle of G must begin with at least one a_s, where $a_s \in A_s$, the set of all source actors, and a cycle must end with at least one a_{act}, where $a_{act} \in A_{act}$, the set of all sink actors
- All actors in G must fire once before starting a new cycle

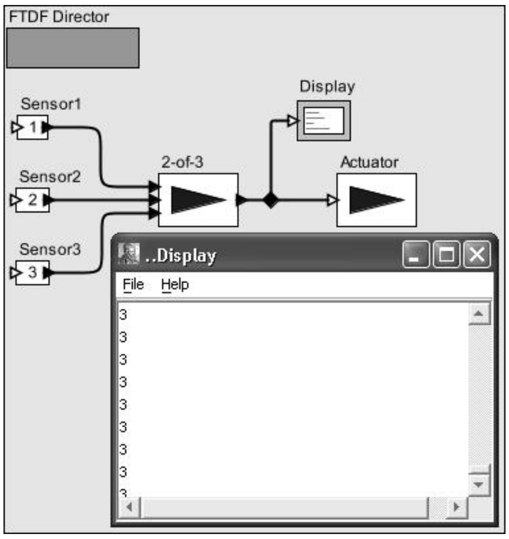
FTDF actors exchange tokens on each cycle under synchronous semantics [7], according to the last rule. The other rules simply require a model to begin with a source actor and end with a sink actor. The FTDF graph must not contain any cycles, either. Based on these constraints on construction of an FTDF graph, the data dependencies of the actors in the graph can be determined. This information can be used to construct a schedule that knows the order of execution of actors in the FTDF graph.

The information presented in this section described the structure and semantics of FTDF. It describes what an FTDF graph does, how it is structured, and allows the reader to gain insight into how this should be implemented. In the following section, the implementation of FTDF is described using the Ptolemy II framework.

4 Implementation in Ptolemy II and Results

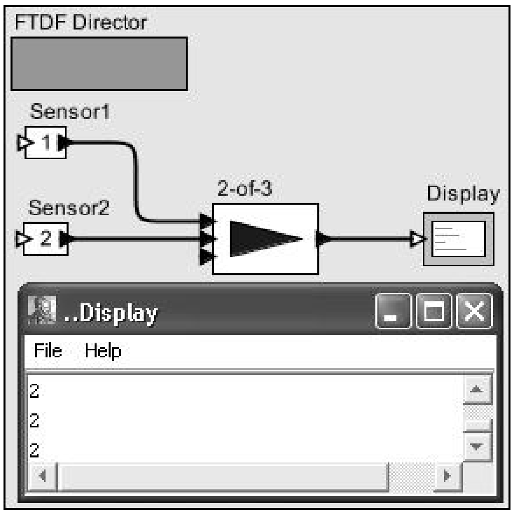
This section highlights the implementation of the FTDF domain in Ptolemy II design environment [5]. Constraints on constructing a legal FTDF graph are discussed in the previous section in terms of composition rules. The implementation of a FTDF model in Ptolemy when constructing a model is guided by constraints on graph construction. If the FTDF graph is legal, and for each actor in the model, a firing function and firing rules are specified, then a FTDF model can be constructed and executed in bounded memory and with no deadlocks.

Ptolemy II offers a rich environment and underlying semantics that allows ease of integrating new models of computation. The approach taken to extend Ptolemy II to support the FTDF domain is to extend the SSDF domain (known as the SDF domain in Ptolemy II) and restrict some semantics of that domain to leverage the implementation of the FTDF domain. An advantage of extending the SSDF domain is that much of the underlying software architecture supports the semantics of FTDF. This reduces the amount of code that must be implemented to create an executable FTDF domain. Also, a rich set of regular actors, or domain polymorphic actors in Ptolemy II, can be used. Input type actors can be implemented in the FTDF domain in Ptolemy II. The functions can be any function that executes a sequential block of code. Examples functions that are relevant to this domain might be deterministic or non-deterministic merges, averages, sums, etc. The functions to implement are up to the model designer, FTDF domain simply supports the semantics discussed in the previous sections. Figures 2 and 3 give an example model that is created in Ptolemy II under the FTDF domain that utilizes an input type actor.



```
3
3
3
3
3
3
3
3
```

Figure 2: This model is a simple example of an *input* actor with three inputs. The firing rules of the input actor, *2-of-3* are the same as the actor in Figure 1. The function of the actor is to simply take the last token that is present on the input port and place on the output port. The model executes with no problems when all inputs are available.



```
2
2
2
```

Figure 3: This model is a simple example of an *input* actor with three inputs, the same model as in Figure 2. Here, it is shown that without one of the inputs, the actor can still produce an output. However, note that if two of the three inputs were not available, the model throws an exception.

In Figure 2, the function of the *2-of-3* actor is to check each input port in order from the top input to the bottom input, and the output is the last token that was present. As shown in the display, the last token was from the bottom port, input port 3. In Figure 3, the function is the same as in Figure 2. Here, it is shown that with the missing input, the actor still is able to fire. In the absence of two of the three signals, the FTDF Director would throw an exception.

The operational semantics of FTDF offers a simple constraint-based execution of a FTDF model. So, for example, the first three composition rules are checked before the model is executed during the FTDF Director's *preinitialize()* phase. The *preinitialize()* phase also constructs the schedule for a FTDF model. If any composition rules are violated, an exception is thrown. It becomes the job of the model designer to correct the problem before the model can be executed.

The SSDF scheduler assumes static production and consumption rates of actors in a model. In FTDF, the SSDF scheduler is replaced by a scheduler that schedules actors in a model based on a topological sort. The sort yields an order of execution of actors from source actors to sink actors such that the data is available at the input ports of downstream actors. This allows for actors to be scheduled in a single thread of execution, and a non-blocking communication style for the domain. Therefore, if a token is not available at an input, it is assumed that a fault occurred with the upstream actor that delivers a token to that input port. Based on the actor's firing function and firing rules, the actor can fire or not fire. In the case that an actor does not have the required number of tokens available at its inputs to fire, an exception is thrown by the FTDF Ptolemy II Director. Under these restrictions, an actor does not have to wait for out-of-order tokens from upstream actors. Furthermore, the last composition rule of a FTDF model only executes each actor in the model once in a cycle. This constraint is imposed in the scheduler as the actors are executed. So, unlike the SSDF domain in Ptolemy II, the model designer is not allowed to change the rate of production of an actor. As a result, only one token is produced per output port of each actor in a FTDF model, and similarly, the downstream actor consumes only one token. This type of schedule is called a *homogenous* schedule [2] in the SSDF domain. Allowing a homogenous schedule restricts the designer to single-rate actors. However, the scheduler is able to compute an admissible schedule that may be executed in bounded memory, in fact, using single place buffers. It also guarantees a deadlock-free execution since the actors communicate using non-blocking reading and writing mechanisms.

5 Conclusions and Future Work

FTDF has been described as a model of computation for modeling fault-tolerance in safety critical, real-time feedback systems. If a legal FTDF graph is constructed that adheres to constraints imposed by the operational semantics of a FTDF graph, then the model can be executed in bounded memory and with no deadlocks. A simplified description of how the FTDF domain is implemented in the Ptolemy II framework is discussed and examples are offered to demonstrate. Implementing FTDF in the Ptolemy II design environment as an extension of SSDF domain leverages the capabilities and components that may be used in the FTDF domain and reduces the amount of code to implement.

For future work, one assumption made in the development of the FTDF domain is that actors can only produce and consume one token per arc on each invocation of an actor during a single cycle. This assumption may be relaxed to handle multi-rate actors. In practical fault-tolerant systems, software is mapped to hardware and possibly either hardware and/or software components may be replicated in a model. This suggests the notion of multiple “processing elements”. In this paper, it is assumed that only one processor is in the model, thus each actor can be considered as being mapped to a single processor. However, future work may include introducing a processing element as a special type of actor in the FTDF domain. This will allow multiple processing elements in a model. Additional work may consist of integrating FTDF domain in Ptolemy II with other design environments and tools, such as Metropolis or a tool constructing fault trees based on the topology of the FTDF model.

6 References

[1] C. Pinello, L. P. Carloni, and A. L. Sangiovanni-Vincentelli. "Fault-tolerant deployment of embedded software for cost-sensitive real-time feedback-control applications," *Proc. Conf. Design, Automation, and Test in Europe (DATE)*, 2004.

[2] E. A. Lee and D. G. Messerschmitt. "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. on Computers*, January, 2004.

[3] E. A. Lee and D. G. Messerschmitt. "Synchronous data flow," *Proc. of the IEEE*, vol. 75, no. 9, September, 1987.

[4] The Metropolis Project Team. "The Metropolis Meta Model Version 0.4," *Technical Report UCB/ERL M04/38*, University of California, Berkeley, CA USA 94720, September, 2004.

[5] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng, "Heterogeneous concurrent Modeling and design in Java (Volume 1: Introduction to Ptolemy II)," *Technical Memorandum UCB/ERL M04/27*, University of California, Berkeley, CA USA 94720, July, 2004.

[6] E. A. Lee and S. Neuendorffer. "Classes and Subclasses in Actor Oriented Designs," *Proc. of the Conference on Formal Methods and Models for Codesign (MEMOCODE)*, June, 2004.

[7] A. Beneviste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The Synchronous Language Twelve Years Later,” *In Proc. of the IEEE*, March 1997.