Model-Based Synthesis and Analysis of Fault Tolerant Data Flow Models

Mark Lee McKelvin Jr

January 30, 2006

Model-Based Synthesis and Analysis of Fault Tolerant Data Flow Models

by

Mark Lee McKelvin, Jr.

Bachelor of Science in Engineering (Clark Atlanta University, Atlanta, Georgia) 2001

A thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science
in

Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Alberto Sangiovanni-Vincentelli, Chair
Professor Jan Rabaey

Spring 2005

Model-Based Synthesis and Analysis of Fault Tolerant Data Flow Models

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Designing embedded software for safety-critical, real-time feedback control applications is a complex and error prone task. The challenges derive from design and implementation costs, time-to-market pressures, late detection of errors introduced early into the design, manual implementation of software, and the need for hardware to test the software (using embedded target platforms or prototypes), amongst other problems. Typical safety-critical applications, like steer-by-wire vehicles, contain a model of the components computing control laws and interacting with a plant using sensors and actuators, as illustrated in Figure 1.1. The control laws are implemented on an execution platform composed of a software layer (i.e. middleware, operating system, etc.) and a hardware layer (i.e. processing elements, communication channels, etc.). A model, or *model of computation (MoC)*, is a mathematical abstraction that explains or predicts the behavior of a physical artifact or phenomenon. Well-defined mathematical models are often useful in the design of such systems because they allow the use of formal validation and analysis techniques, and they reduce ambiguity in capturing design specifications and translating specifications to an implementation, or deployment. The model is often times created in a visual, interactive environment using block diagrams or flowcharts, however visual representation is not necessary. The benefits of a model-based design are that models may be reused in various steps in the design and development of systems, they may be simulated to validate or clarify the behavior, they may be debugged before software is written, and they help facilitate automatic code generation which aid to reduce coding errors and coding time. Furthermore, formal analysis techniques may be used to quantitatively analyze models.

Fault tolerance is an important aspect of safety. It is the ability for a system to behave correctly in the presence of component failures, and it is commonly used in safety-critical applications. The

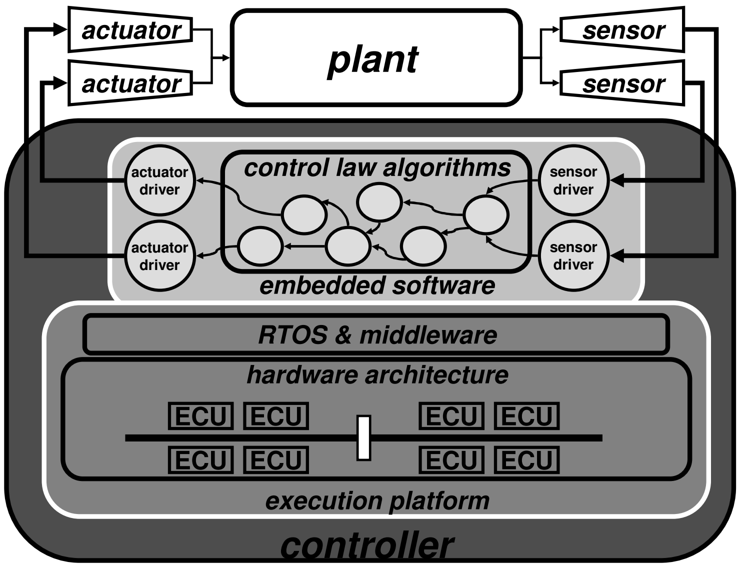


Figure 1.1. An example feedback control system composed of a plant and an execution platform implementing control algorithms.

aerospace and automotive industries offer numerous examples of such applications. A number of methods are used in various domains to achieve fault tolerance. Some methods include validation, time redundancy (performing the same operations a number of times and evaluating the results), or error correction (information redundancy). An additional method to achieving fault tolerance is duplicating hardware and software components, a solution that is often more expensive than needed. In safety-critical applications such as automotive electronics, a subset of the functionalities must be guaranteed while other functionalities are not critical to the safe operation of a vehicle. In this case, the designers must make sure that a critical subset is operational under the potential faults of the architecture (i.e. using hardware and software redundancy).

1.1 Motivation

A model of computation called Fault Tolerant Data Flow (FTDF) was recently introduced in a proposed interactive design flow [21] to describe, at the highest level of abstraction, the design of fault tolerant requirements on the functionality of a system. One of the key parts of the design flow is a mapping of functionality onto a chosen architecture under a set of constraints. The mapping takes as input the behavior of the system represented by FTDF, the architecture for which the application is mapped onto, and a set of constraints which satisfies the design requirements. The result of the mapping process is a schedule of FTDF components that are mapped to the given architecture satisfying the input constraints. From this mapped schedule, a series of analysis steps may be taken to further refine the schedule before deploying the system onto a target platform that may be used for rapid prototyping or a real implementation. The analysis techniques allow the designer to evaluate the mapped schedule before deployment to determine the system cost or measure the amount of fault coverage achieved. The analysis capabilities and software synthesis from a mapped schedule to an implementation allow the designer to quickly and efficiently explore the design space by hinting on changes that may be made on the functionality, architecture, or mapping process, and perhaps derive a better fault coverage or achieve a lower implementation cost. A complete design flow for this kind of application must maintain a consistent model at every step, from specification to deployment. A single model that captures the requirements and used for implementing synthesis (i.e. code generation, fault trees for fault tree analysis, etc.) and analysis (i.e. timing analysis, reliability analysis, etc.) techniques, can alleviate many of the problems of designing complex systems, which are typically done by a team of designers. The overall synthesis-based design flow is discussed further in Chapter 2.

This work is motivated by the following scenario, where a designer wishes to rapidly prototype a system using FTDF. The designer could take the results of a mapping such that the constraints to that mapping process are satisfied, and then implement an execution on a target platform. But, the designer has no way of evaluating how good the fault coverage or cost of implementation happens to be. The addition of a fault tree synthesis tool in the design flow will allow the designer to not only implement a system that satisfies input constraints, but the designer could evaluate a system that also meets specified dependability requirements. The information from a dependability analysis could tell the designer the difference in cost or fault coverage for multiple system mappings. Then, using the same system model which produced the fault tree for dependability analysis, the designer could proceed (if satisfied with results of the analysis) to deploy the model on a target system for rapid prototyping. This process can be done quicker and more efficient than if the analysis and deployment are manual. On the other hand, the designer may not have the necessary information to make a reasonable assumption regarding the behavior or timing requirements of the system under design. The designer could quickly generate a prototype of the system with little effort, and possibly use the timing information from the prototype to specify better timing constraints or validate the correct behavior of the system when faults are injected (or encountered from normal operation). A software synthesis process that synthesizes a FTDF model to executable code and fault tree synthesis process that generates a fault tree for analysis adds additional capabilities to a synthesis-based design flow that can be completely automated.

This thesis enhances and complements the work presented in [21] for a synthesis-based design methodology in two important aspects. First, it addresses a method that facilitates code generation from a FTDF model, including the synchronization, communication, and redundancy management mechanisms required to deploy the system on a distributed architecture. The deployment gives an early feedback of the system under development from an executable simulation on a host or on a target platform that is representative of the final target system. Second, it presents a formal approach to analyzing dependability of a system. The approach is based on the automatic synthesis of a fault tree from a mapped FTDF model. The fault tree allows the use of existing fault-tree analysis tools, such as the Item Toolkit [7], to analyze the deployments and score them on a dependability dimension. The tools described in this paper are already integrated into the synthesis-based design flow described in Chapter 2 and shown in Figure 2.3. As a result of these new additions, designers can quickly evaluate the dependability of multiple candidate systems, and when the designer is satisfied with the quality of the design, the system can quickly reach a deployment phase using automatic code generation. Furthermore, automatic code generation enables rapid prototyping for design space exploration. These two new aspects find their natural positioning in the overall design flow described in Chapter 2, however, they can be valuable design and analysis tools independent of the automatic mapping phase described in [21]. If designers are interested in describing the redundant (i.e. using replica software and multiple hardware components) FTDF mapping manually by bypassing the automatic mapping phase, they can still take advantage of the automatic code generation, fault tree synthesis and analysis. These two aspects further complement one another since the analysis capability of the design flow provides a method to analyze the structure of a system, whereas the deployment resulting from code generation provides the mechanism to analyze or validate the behavior of the system. Both aspects enable design exploration by providing feedback to the designer.

1.2 Organization of Report

The remainder of this report is organized as follows. Chapter 2 reviews the proposed synthesis-based design methodology for which this work integrates into. Chapter 3 gives an overview of the FTDF MoC. Chapter 4 discusses the synthesis of implementation code from a FTDF model. In that chapter, the runtime platform for a FTDF implementation is described, and a modeling environment for constructing a FTDF application that is executed by the runtime platform. Chapter 5 discusses a formal description of the fault tree analysis technique which includes a fault tree synthesis algorithm. In Chapter 6 results from two examples are presented to illustrate software synthesis and fault tree analysis capabilities from a mapped FTDF model. Chapter 7 summarizes this paper with conclusions and a brief discussion of future work.

Chapter 2

Synthesis-Based Design Methodology

A synthesis-based design methodology is proposed that involves the designer in an interactive design method for exploring the redundancy versus cost trade-off of safety-critical systems specified using the FTDF MoC. *Software replication* is used to achieve fault tolerance: critical routines are replicated statically (at compile time) and executed on separate *electronic control units (ECUs)*, and the processed data is routed on multiple communication paths to withstand channel failures. For the sake of simplicity, the fault model used in this paper is *fail silence* [11]. Fail silence assumes that components either provide correct results or do not provide any result at all. Other common fault models might include value errors or Byzantine fault models.

The design methodology generates a valid fault-tolerant deployment of a feedback control system given as a FTDF graph by specifying the *fault-tolerant binding.* A fault-tolerant binding ensures that for each fault scenario, or *failure pattern* [5], the execution of a corresponding subset of the actors in the FTDF graph, $\mathcal{G}$ must be guaranteed. Figure 2.1 illustrates a FTDF graph for a paradigmatic feedback-control application, the inverted pendulum control system. In the example, the controller is described as a bipartite directed graph $\mathcal{G}$ where the vertices, called actors and communication media, represent software processes and data communication. In addition, Figure 2.2 is a possible *platform graph*, *PG*, where vertices represent ECUs and communication channels and edges describe their interconnections. The example repeats the following sequence at each period $T_{\max}$: (1) sensors are sampled, (2) software routines are executed, and (3) actuators are updated with the newly-processed data. In order to guarantee correct operation, the worst-case execution time (WCET) among all possible iterations must be smaller than the given period, $T_{\max}$ (a real-time constraint for this example). Moreover, the control algorithms must be executed in

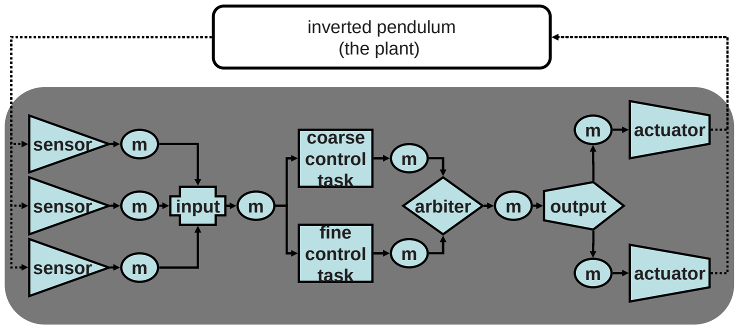


Figure 2.1. Controlling an inverted pendulum example.

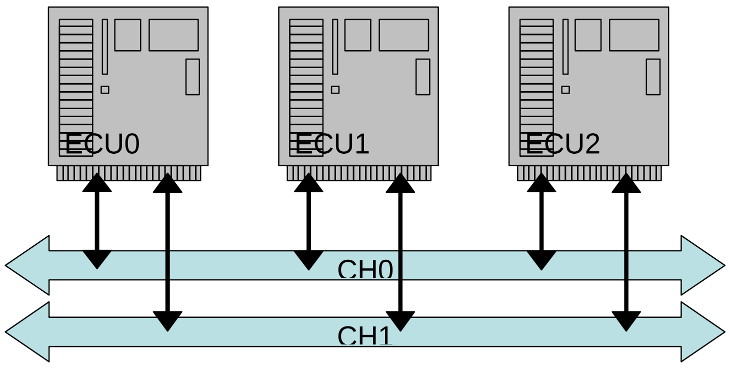


Figure 2.2. An example of a simple platform graph.

spite of the possible platform faults. Figure 2.3 illustrates the proposed interactive design flow where designers

- specify the controller (the top-left FTDF graph);
- assemble the execution platform (the top-right PG);
- specify a set of failure patterns (subsets of PG);
- specify the fault tolerance binding (fault behavior);

A synthesis tool automatically derives a fault-tolerant schedule as a result of the mapping in Figure 2.3 that uses a timing analysis engine and other analysis techniques in a verification tool to check whether the fault-tolerant behavior and the timing constraints are met. Dependability analysis adds another dimension to the analysis capabilities of this flow. For example, if more than one solution is found, one would like to assess precisely their dependability to correctly explore the fault-tolerance versus cost trade-off. Once the designer is satisfied with the analysis and accepts the metrics produced, then a deployment via code generation may take place. One might use the deployment to further evaluate the system design.

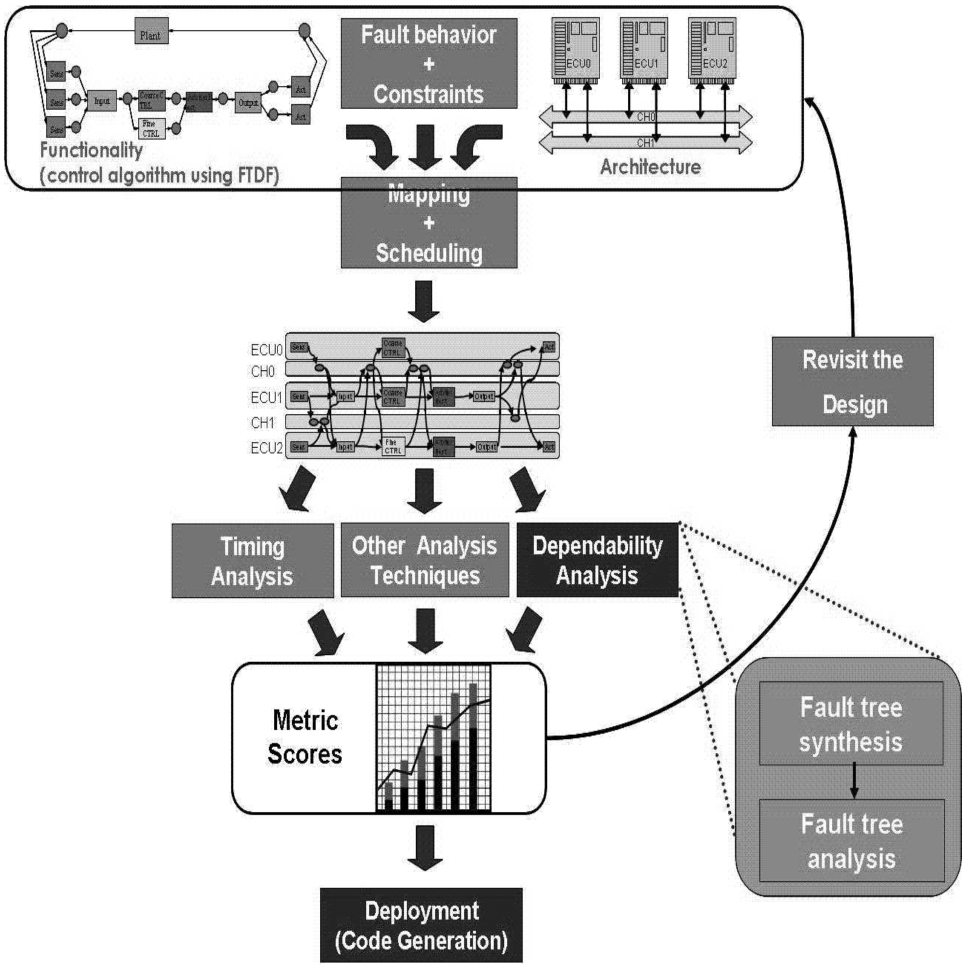


Figure 2.3. Proposed design flow.

Chapter 3

Fault Tolerant Data Flow (FTDF)

Fault Tolerant Data Flow is a synchronous [4] model of computation: every actor executes once per iteration, satisfying the precedence order dictated by the data dependencies. Then the next iteration may start. FTDF is designed to address the specification of fault-tolerance in safety-critical, real-time feedback control systems. Furthermore, FTDF is a data flow [16] [14] variant that targets efficient periodic executions and alleviates ambiguous specifications in the design, modeling, analysis, and validation of safety-critical feedback control systems. The structure of FTDF enables formal analysis and automatic synthesis tools and techniques. This section reviews the fundamental components of an FTDF model: tokens, actors, and communication media. An FTDF graph structure provides the structural dependencies amongst components in a FTDF model.

3.1 Tokens

Tokens are encapsulations of data. In addition, in the FTDF domain, tokens are appended with two fields: a *valid* field and an *epoch* field. The valid field is used to record the Boolean outcome of some fault-detection algorithm (e.g. majority voting, checksum, CRC) since FTDF is designed to be fault model independent. Moreover, an actor may explicitly mark any of its output tokens as invalid to inform downstream actors of some error. An actor receiving a token can check the token's validity before attempting to use it. The epoch field is used in an execution as a synchronization mechanism for the distributed processes.

3.2 Actors

In an actor-oriented design framework [15], actors are functional components that execute and communicate with other actors in a model. An actor contains ports that are connected via an abstraction of communication channels. In FTDF, this abstraction is referred to as a *medium*. Actors also contain a *firing rule* and a *firing function* to specify the behavior of an actor. A firing rule is a guard condition that must be satisfied by input signals to the actor. The firing function executes a body code that implements a particular functionality of the actor.

Formally, an actor may be defined as a data flow process [16] where: $F : S^n \rightarrow S^m$, 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, $f : S^n \rightarrow S^m$, 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 fixed point [6] 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.

In FTDF, actors are typed. A FTDF actor belongs to one of six types. *Sensor* and *Actuator* actors read and update respectively the sensor and actuator devices interacting with the plant. *Input* actors perform sensor fusion. *Output* actors are used to balance the load on the actuators, while *Task* actors are responsible for the computation workload. *Arbiter* actors mix the values that come from actors with different criticality to reach to the same output actor. The set of firing rules for Sensors, Actuators, Tasks, and Output actors prescribes that the actor can fire only when all incoming signals (tokens) are present. This is represented with the following notation $U = \{(*, *, ..., *)\}$, where the symbol "$*$" represents the presence of a value, hence the actor can fire only if all incoming tokens are present[1]. This is the typical firing rule found in other data flow languages [14] [16]. On the other hand, the input and arbiter actors can fire on the presence of a subset of its inputs. For example, an input actor that may fire if at least two of three input tokens are available would have the following set of firing rules:

$$U = \{(*, *, *), (\perp, *, *), (*, \perp, *), (*, *, \perp)\}, \tag{3.1}$$

[1]Sensor actors have no inputs, so they can always fire, at the beginning of each period.

where $\perp$ denotes the absence of a value. The execution semantics of FTDF dictates that, in the absence of faults, an actor should wait for all of its input tokens before firing. When it is detected that a token is missing for a given period, then the partial firing rules may still enable the actor to fire. In addition to the six types of actors described above, *State Memory* may be present in a FTDF graph. This component stores results produced during the current period of a FTDF execution for use in the following period.

3.3 Communication Media

Communication occurs via unidirectional communication media. All replicas of the *same* source actor write to the same medium, and all destination actors read from it. Media act both as mergers and as repeaters sending the single "merged" result to all destinations. In general, the medium provides the correct merged result or an invalid token if no correct result is determined. More specifically, the behavior of a media, as with the actors, is based on the fault model chosen. The fault model restricts the number of faults and their types. A number of fault models are possible. Depending on the fault model, the communication media may have a different behavior. Assuming fail-silence, merging amounts to selecting *any* of the valid results because fail silence assumes components produce correct results or no results at all; assuming value errors, majority voting is necessary to do the merge; and assuming Byzantine faults, multiple rounds of voting is needed in order to merge the data (see the consensus problem [3]). Since fail silence is assumed for communication media (and actors) throughout this paper, it suggests that no voting is necessary for the communication media, and the media will produce results or none at all. Communication media must be distributed to withstand platform faults, typically this means having a repeater on each source ECU and a merger on each destination ECU (using broadcasting communication channels helps reduce message traffic greatly) in an implementation of a FTDF model.

3.4 Composition Rules

The following rules specify the set of valid actor compositions to obtain a legal FTDF graph. Some basic rules (e.g. all input and output ports of an actor should be connected, data-types should be matched, etc.) are common to most dataflow models and are assumed implicitly here. A FTDF graph $\mathcal{G}$ is a pair (V, E) where $V = A \cup M$ is the set of vertices and $E \subset (A \times M) \cup (M \times A)$

is the set of directed edges. A set of actors is given by A

$$A = A_S \cup A_{Act} \cup A_I \cup A_O \cup A_T \cup A_A \cup A_M, \tag{3.2}$$

where A is composed of a partition of the six types of actors and the state memory actors. M is a set of communication media. Then, a FTDF graph $\mathcal{G}$ is *legal* if the following holds:

- $\mathcal{G}$ contains no causality cycles,
- the source to *Input* actors are *Sensor* actors, and the outputs of *Sensor* actors are *Input* actors,
- the source to *Actuator* actors are *Output* actors, and the outputs of *Output* actors are *Actuator* actors, and
- *Sensor* actors have no inputs from other actors, and *Actuator* actors have no outputs to any other actor.

Finally FTDF graphs can express redundancy, i.e. one or more actors may be replicated. All the replicas of an actor $v \in A$ are denoted by $\mathcal{R}(v) \subset A$. Note that any two actors in $\mathcal{R}(v)$ are of the same type and must compute the same function.

Chapter 4

Software Synthesis of FTDF Models

This chapter introduces the method for which a FTDF application is synthesized into executable code from a FTDF model. The synthesis is performed in two steps. First, a runtime library is constructed. The runtime library is debugged, portable code that can be used to implement a mapped FTDF application on a target platform [1]. It is hand coded and compiled once, but it can be modified as specifications change. The target platform can be a real-time embedded system or personal computer (PC) with a supporting operating system for distributed application development. Secondly, the runtime platform must be configured to execute a specific FTDF application. The configuration defines the necessary data structures and appropriate actor connections for a FTDF application. The configuration file specifies proper dependency connections between actor components based on the FTDF model structure. It also translates actor annotations and parameters, such as firing rules and firing functions to the runtime platform. Figure 4.1 depicts the flow in terms of the files used in the code generation process. It proceeds from modeling FTDF applications in a modeling environment and creating an executable application from a compilation of the application files with a runtime library.

4.1 Background

The process of manually implementing an application on the runtime platform can become tedious, time consuming, and error prone due to challenges with redundancy management and synchronization of FTDF components. Automatic code generation enables design exploration through

[1]Mapped FTDF refers to a mapped FTDF schedule which is the result of a functionality to architecture mapping process as described in Chapter 2.

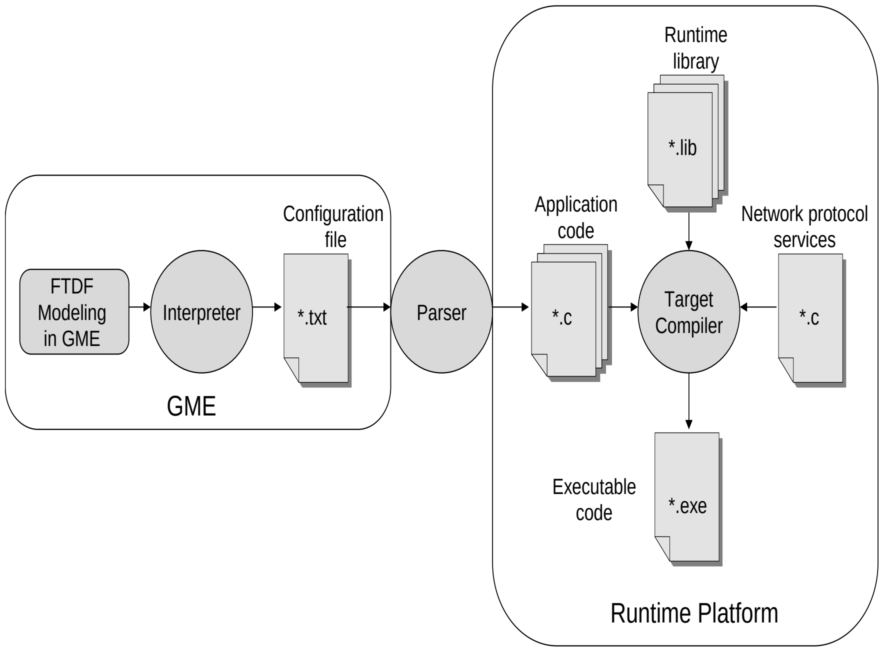


Figure 4.1. A flow diagram of the software synthesis method.

the use of executable models that allow the designer to quickly prototype a mapped FTDF application on a target platform or execute a simulation on a single PC. A domain specific environment is used to generate a configuration file that interfaces with the runtime platform from a visual FTDF model. The configuration file specifies dependency connections between actor components based on the FTDF model structure. It also translates actor annotations and parameters, such as firing rules and firing functions to the runtime platform. The method from which a visual domain specific modeling environment is used to model and interface with the runtime platform using the Generic Modeling Environment (GME) is addressed in [22]. The advantage of the two step synthesis process is that the runtime platform closes the semantics gap between FTDF modeling and its implementation on an execution platform. Once the semantic gap is small enough, it is relatively easier to generate a correct configuration for the runtime platform. Furthermore, if a new target is chosen for the execution platform, only layers within the runtime platform that are affected are changed. The methodology is reviewed in this section.

4.2 Related Work

A number of tools have been used for code generation from mathematical models. These tools, along with the approach presented in this section, translate mathematical models into programs written in a system-level programming language such as C++ or C. In [9], C++ code is automatically generated from a hybrid system modeling language called CHARON. The strategy consists of a two step procedure where primitives of the hybrid automaton are translated into equivalent implementations, followed by a scheduling of primitive components according to the data dependency in the model. Other tools include, MathWorks Simulink [17], a commonly used commercial tool that supports code generation from data flow block diagrams, and Esterel Technologies Esterel Studio [28], which generates Verilog/VHDL code.

4.3 Runtime Platform

The *runtime platform* is an executable program that supports the execution and deployment of a distributed FTDF application. Applications in the domain of safety-critical control systems are expected to execute periodically on a possibly infinite stream of data on a computer-based system within bounded memory. The system is distributed across a set of processors on a target system, like a distributed embedded real-time system or a distribution of personal computers. The runtime platform can be used for simulation, or it can be used to prototype an implementation if the target system supports directly the runtime platform.

The runtime platform is a library of debugged ANSI C [8] code that creates necessary data structures for coordinating the communication and computation of a network of FTDF components on a target platform. The target platform chosen for this project is a network of personal computers running the Linux operating system communicating with an Internet Protocol (IP) stack. The runtime platform enables rapid prototyping and simulation on various systems that can be networked using the IP protocol suite. Ideally, the designer could choose the target platform to execute an application, for example using a real-time operating system instead of Linux.

Conceptually, the architecture of the runtime platform is composed of three major components: the *scheduler*, the *communication network interface*, and the *watchdog interface*. Figure 4.2 illustrates the architecture of the runtime platform. The scheduler executes a periodic schedule of actors in the FTDF application. The communication network interface manages network communications using a client/server communication model. The watchdog interface supports a timeout mechanism in the runtime platform. Each component is discussed further below.

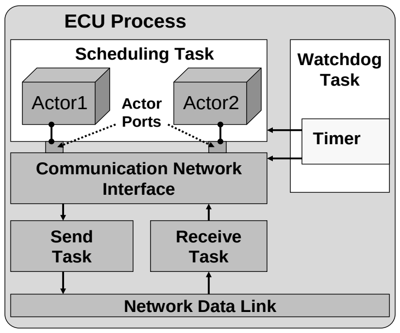


Figure 4.2. Architecture of the runtime platform.

4.3.1 Data Structures

This section gives a brief overview of some of the key data structures in the runtime environment. The data structures implement the main components of a FTDF model: *Tokens* and *Actors*. They are composed of a number of parameters that describe the operational semantics of a FTDF model and support the implementation as executable code.

Tokens

As mentioned in Section 3.1, tokens are encapsulations of data exchanged between actors. A data structure used by the runtime platform for a token is given in the code snippet below.

```
typedef struct token {
  int epoch;            //Tokens iteration number
  bool valid;           //Validity field of token
  int dsz;              //Size of data in data field
  dtype *data;          //Data
};
```

The communication network interface transports the tokens between actors. The token data structure contains four fields: *epoch*, an integer value representing the iteration number; *valid* field, a boolean value that serves to mark the result of an error detection technique; and *dsz*, size of data in the *data* field. Token data structures are accessible to the user through the actor firing rule and firing function. Within those two functions, the user can read or write tokens from pre-allocated memory locations called buffers. Buffers are memory storage locations for tokens, as defined in the actor data structures. Buffers are interfaces between the application code and the communication network interface.

Actors

Actor objects are defined in the following data structure.

```
typedef struct actor {
  char name[NAMESZ+1];       //User-defined name of actor in FTDF network
  int epoch;                 //Actor's current epoch
  bool (*fire) (struct actor *);     //User defined firing rule
  void (*execute) (struct actor *);  //User defined execution function
  int critical;         //Criticality level
  int timeout;          //Firing rule timeout (millisecs)
  actorPort *port;      //ptr to actorPort stuct
};
```

There are a number of parameters that describe the various types of actor instances.

- **name**: a user-defined identifier composed of a series of characters.
- **epoch**: an integer valued iteration number of an actor instance. This field is useful because the epoch imposes an ordering on tokens in the application. The ordering is imposed by comparing the actor's epoch to the epoch of input tokens. Before a token is written to the output buffer, the epoch field is updated with the most recent epoch out of all its input tokens.
- **fire**: a user-defined function that is used as the firing rule for an actor. The function is specified in a well-defined interface that allows flexibility in implementing an arbitrary firing rule. The user must have a function header that takes a pointer to an actor and returns a boolean value that represents the result of the firing rule during a single iteration. An example is given in **actorFire** code:

  ```
  bool actorFire (actor *a) {

  //Firing rule for Arbiter actor.  It fires
  //once it receives at least one valid input
  //on time with an epoch that is greater
  //than the current local actor epoch.

    int i = 0;
    int validCt = 0; //valid tokens in buffer count
    token temp;
  ```

```
  for (i = 0; i < 2; i++) {  //buf with 2 inputs
    temp = readToken(a->port->inBuf+i);
    if ((temp.epoch > a->epoch) &&
      (temp.valid == TRUE)) {
      validCt++;
    } else {
      //clear token in buf and wait for
      //a valid token to come
      resetToken(a->port->inBuf+i);
    }
  }
  if (validCt >= 1) return TRUE;
  return FALSE;
}
```

The user should use **readToken** to read tokens from the input buffer of the actor for the current iteration. The function **readToken** is defined in the runtime platform library of functions, and its purpose is to perform a blocking read to ensure proper data synchronization. Similarly, **resetToken** may be used to remove tokens from the input buffers. The **fire** function will be invoked by the scheduler as many times as possible until the actor either times out or the firing rule is satisfied. So, the firing rule should contain no infinite loops or it will hang the runtime platform during execution.

- **execute**: a user-defined firing function for the actor. It serves as the key functionality of an actor. The function is specified by the user in a well-defined interface. An example of a user-defined firing function is given below.

```
void a4Exec (actor *a) {

  token temp;
  int i = 0;

  //Read valid input tokens
  do {
    temp = readToken(a->port->inBuf+i);
    i++;
  } while (temp.valid != TRUE);

  //Do some work on the tokens data. This is
  //the main execution of this actor.
  temp.data[0] = temp.data[0] * 2;

  //write to output - expects 1 output trigger
  //with updated epoch
  temp.epoch = a->epoch;
  temp.valid = TRUE;
  writeToken (&temp,a->port->outBuf+0);
}
```

The function header must be defined by accepting a pointer to an actor structure and returning nothing, i.e. *void functionExecute (actor *a).* The function name is arbitrary. Also, the user should use **readToken** and **writeToken** library calls to read and write tokens within the firing function. The two functions implement blocking read and write calls, respectively, to ensure data synchronization. Tokens may be manipulated in the firing function to compute a function of the input tokens.

- **critical**: an integer value specifying the criticality level of an actor. This is a user-specified parameter. In FTDF, the criticality level is used to determine the level of service an actor provides. For example, actors that are more critical than others should provide more reliable service than less critical actors. To ensure an actor meets this guarantee depends on how well the designer designs the control algorithm and the amount of redundancy added to the design.
- **timeout**: a user-specified parameter that provides an upper bound on the time for an actor to wait on input tokens. A situation might occur where an actor is blocked because it is awaiting input tokens. The timeout value is used by a timer in the runtime platform to discontinue waiting if the timeout value is reached by stopping the blocked process. This behavior prevents the scheduler from blocking infinitely. The granularity of the time value is determined by the accuracy of a timing mechanism provided by the target platform since target platforms support various timing accuracy.
- **port**: an actor port is a data structure that contains input and output buffers for the actor. Actors with replica source actors have a one-place buffer for each group of replicated sources. Since FTDF is synchronous and assumes actors are single-rate, one-place buffers are used to fix the memory requirements. As a result, the memory is bound and deterministic. Furthermore, an assumption is that the replicated sources execute the exact same firing function code, hence they produce the same output tokens. The actor ports are used by the runtime environment to send and receive tokens between the FTDF application and the communication network interface. The memory locations are determined by the designer before execution, thus no dynamic memory is used.

4.3.2 Scheduler

The scheduler is implemented as a single thread in the runtime platform. Its purpose is to execute actors in a topological order. The order of actor executions is determined by the structure of the mapped FTDF graph that is under execution in the platform. The user must provide the runtime platform with a global list of actors in the FTDF application. The user assign the appropriate firing rule and firing functions, manually place the actors in topological order, specify the firing rule and firing function of each actor, and also specify the memory locations of each actor's input and output buffers. The order of actors should be from *Sensor* actors (placed at the head of a global list of all actors in the application) to the *Actuator* actors. Each actor executes once per *iteration* according to FTDF semantics. An iteration is an invocation of each actor once in a single cycle under the assumption that actors are single-rate (produce one token per execution). The scheduler repeatedly calls actor firing rules and firing functions in an infinite loop. The code snippet below implements the periodic, static schedule.

```
while (1) {  //Infinite loop of schedule
  printf ("\nNew iteration-------------------\n");
  for (i = 0; i < nActors; i++) {  //For each actor
  a = actorList[i]; //Get next actor to execute

  //Run actors firing rule
  awaitFire(a);

  //Set local actor's current epoch
  a->epoch = getMaxEpoch(a->port->ins, a->port->inBuf);

  //Execute actors function
  runExec(a);

  //Send tokens from output buffers over Communication
  //Network Interface
  sendTokens(a);
  }
}
```

When the scheduler begins, it first takes the next actor from the global list, **actorList**. It then executes a function called **awaitFire** that calls the firing rule of actor a, as in the figure. The **awaitFire** function is a blocking call that reads the input buffers of the actor, and the function will return when either the timer expires on the timer task or the actor receives enough valid tokens to satisfy the firing rule. After the firing rule returns, the scheduler looks at each input token and determine the maximum epoch of that group of tokens. The maximum epoch value tells the scheduler the current iteration of the scheduler so the actor may be synchronized with the latest epoch of the distributed application. The epoch in the token is used to establish a global ordering since some processes may be distributed over the network and execute at different rates. The main function of the actor is executed in function **runExec**. This function simply calls the user-defined firing function of the actor. Finally, the output tokens of the actor are sent to the communication network interface using the **sendTokens** function.

4.3.3 Communication Network Interface

The communication network interface provides the communication services between actors on the same local memory and between actors distributed on remote processors. The communication network interface is an event-based layer that supports distributed network communication between the FTDF application and the network layer using a client/server communication model [27]. A *client* requests services that are provided by a *server*. Client/server communication models are common in distributed applications. The network layer uses a network stack to implement distributed communication across a network of processes. These processes may be personal computers, or it may implement a virtual network of the FTDF application on one personal computer, or process. The communication network interface is hidden to the user. Currently, the network interface supports the IP protocol suite and interprocess communications. But, in the future, this interface may be extended to support communication mechanisms of specific target architectures.

As mentioned above, the scheduler calls **sendTokens**, a blocking function that sends tokens to the communication network interface. The network interface takes each token in the actor's output buffer and pass the token to either the network interface buffer or to a shared memory location (memory location on the same processor). The network interface buffer is a circular buffer that temporarily stores tokens until the network is ready to send the token across the network. To avoid overwriting tokens, the network interface utilizes blocking read and write functions.

To support different protocols, the network interfaces with the physical network stack via a standard set of functions. The standard interface is a set of functions that define a number of services offered by the network protocol. The services in the data structure below are defined by the user in a separate header file. The definitions in the header file are protocol dependent while the standard set of services in the **services** data structure are protocol independent. This technique of providing modularity and abstracting unnecessary details from the designer is similar to a technique implemented commonly in Linux device driver [24] development.

```
typedef struct services {
```

```
void (*load) (struct services_ *);
int (*servInit) (char *, int);
int (*sendInit) (char *, int);
int (*serv) (int, char *, int);
int (*send) (int, char *, int);
void (*stop) (int); //closes a socket
};
```

The function **load** is used during the initialization phase of the network interface to initialize the services found in the **services** data structure. The initialization tells the network interface where the service functions are found. The current architecture allows the designer to define the services in a separate header file. This modular architecture supports changes to implementation details of the services. For example, if the designer chooses to use Transmission Control Protocol (TCP) over IP, service calls using TCP can be used. In contrast, the designer may decide to use inter-process communication mechanisms via pipes and queues to implement network communications. The function calls supported by both protocols are different. The functions **servInit** and **sendInit** are functions that will start the server and client respectively based on protocol- dependent initialization routines. They are executed only once during network initialization, and since typical network protocols return *file descriptors* that identify a specific connection to the network stack, the **servInit** and **sendInit** functions return a file descriptor for the server and client respectively. File descriptors are handles, or references, to an object or connection. Typically the object is a network socket or file. The **serv** and **send** services are used in the network interface to receive and send data over the network. These functions may be called as many times as needed by the application. Finally, to stop services the function call **stop** is used. It takes as input a descriptor that identifies a connection to stop. An example of using the User Datagram Protocol (UDP) network service calls over the Internet Protocol (IP) is provided in the Appendix.

4.3.4 Watchdog Interface

The watchdog interface is started in a thread, and its job is to monitor synchronization mechanisms that are used in the scheduler and throughout various parts of the runtime platform. POSIX [27] compatible threads and mutual exclusion (mutex) locks are used in the runtime platform. The watchdog interface serves as a container for controlling and manipulating locks and controlling a timer task. One of the signaling service it provides to the scheduler is to monitor the timeout value for the actor as it awaits valid input tokens using its timer task. The timeout task is implemented as part of the watchdog interface, and it may be configured with the appropriate accuracy needed by the application, as long as the timing accuracy is supported by the underlying operating system. The timer task works by starting a timer when the scheduler executes the firing rule for an actor. The firing rule is executed in a thread, and if the time elapse (given by the *timeout* parameter of an actor) before the scheduler signals to the timer task to stop the timer, the timer task will stop the firing rule and signal to the scheduler that a timeout has occurred. The scheduler then responds by skipping the execution of that actor and alerting the user.

Currently, the watchdog interface manages timeouts in the FTDF application by using its timer task. However, similar to the standard services offered by the communication network interface, the watchdog interface is constructed such that the designer can introduce additional services, like a fault injection task that may randomly introduce faults in the runtime platform.

4.4 FTDF Application Modeling

The main flow of the FTDF modeling environment for FTDF applications is presented in Figure 4.1. The flow begins with the designer constructing a FTDF application model in the Generic Modeling Environment (GME). GME stores the model in an internal model database. The designer instantiates an interpreter from the GME user interface to initiate the model interpretation on objects in the model database. The model interpretation translates the application model into a configuration file for the runtime environment. Then, the configuration file is used to configure a FTDF application in the runtime environment. This section begins by providing background information in GME and how a designer would use it to build a domain specific modeling environment for FTDF models. Key features of the FTDF modeling domain are discussed.

4.4.1 Generic Modeling Environment (GME)

The Generic Modeling Environment [1] developed by the Institute for Software Integrated Systems (ISIS) at Vanderbilt University is a generic, configurable tool for constructing *domain-specific modeling environments*. A domain-specific modeling environment is an environment for which domain experts can quickly design and analyze problems in that domain. Some examples include, SPICE [25] for electrical circuit analysis, Mathworks Matlab [17] for numerical computations, and LaTex [13] for typesetting. The configuration of GME for a specific domain is accomplished with metamodels that specify the *modeling paradigm* for which model instances are created. The modeling paradigm is the syntactic, semantic, and presentation information necessary to create models within a particular domain. After construction of the modeling paradigm, model translators called *interpreters* are designed and implemented. The interpreters work with all models of the domain-specific environment to translate a given model to a physical artifact like a source file for an analysis program or execution environment.

Modeling Paradigm Definition

This section describes the steps to constructing a metmodel for a domain by defining the modeling paradigm. One of the first steps to creating a domain-specific environment is to define the modeling paradigm. The modeling paradigm addresses a number of decisions, such as: what is to be modeled, how it is modeled, what information will be stored, and so forth. Typically the end-user does not change the paradigm definition, unless of course, specifications of the paradigm definition change. Example paradigms include domains for modeling signal flow graphs and hardware architectures, or a domain for chemical process modeling that is often found in chemical process engineering.

Once the modeling paradigm is designed, a metamodel that formally describes the modeling paradigm is constructed. The metamodel defines the types of objects that may be used during the modeling process, how the objects appear visually, the attributes associated with the objects, and how the objects relate to one another. In GME, the metamodel is defined using Unified Modeling Language (UML) [20] class diagrams. The metamodel also contains a description of constraints that the modeling environment must enforce on a particular model that is constructed using the metamodel. The Object Constraint Language (OCL) [20] is used to express these constraints using standard predicate logic.

After the metamodel is constructed in GME, it is used by the domain expert to build domain-specific models. Before the metamodel can be used, it is checked for inconsistencies within the UML class diagram structures such that the relationships between objects are defined according to UML syntax. After a successful check, the metamodel automatically produces a finite set of objects that are made available to the domain expert for building domain-specific models. In many cases, the domain expert chooses to refine or update the metamodel. This is done easily in GME by editing the metamodel and generating the set of objects that are used by the domain expert to build models from that domain.

Metamodeling in GME

This section discusses the domain-independent components in GME that are used to build a metamodel. This presents a challenge in modeling for the designer in itself, since the designer must decide how to best represent the domain using a number of primitive modeling components. The components include First Class Objects (FCOs), Aspects, Attributes, and Preferences. There are five components in GME that are designated as FCOs. They are *Models*, *Atoms*, *References*, *Connections*, and *Sets*. The components that are used by the domain designer in GME to construct metamodels are briefly discussed below.

- **Models**

  A *Model* is an abstract object that represents an entity defined by the domain. A Model contains state, behavior, and identity. It can be created and manipulated in GME. A Model can be hierarchical, and in such case, it is referred to as a *compound model*. Compound models contain other objects known as *parts*, and these parts can be other Models or other FCOs. Compound models allow the designer to express various levels of abstraction in the domain. Unlike a compound model, a *primitive model* is one which does not contain any parts. A Model can also be associated with a default icon or user-defined icon for visual representation.

- **Atoms**

  Atoms are elementary parts that do not contain any internal structure, hence, they do not contain parts, but they do have attributes that are defined by the metamodel. Atoms represent indivisible entities that are contained in a parent object (an object that contain parts). Like model FCOs, a user-defined icon may be associated with an Atom.

- **References**

  A *Reference* is an object that refers to another modeling object, like an Atom or Model. In concept, a Reference is similar to a pointer that is found in common programming languages. It is often used to reference a part that has already been created in the domain environment, or it could reference "NULL" to act as a placeholder. A Reference can also be associated with an icon.

- **Connections**

  A *Connection* is an edge in a graph of parts. It is given in GME as a directed or undirected line with a part on each end. It expresses a relationship between two parts. As specified by the semantics of the modeling paradigm, the metamodel can be used to define the type of Connection, parts that may be related through Connections, and a number of other attributes that further restrict the Connection, such as multiplicity.

- **Sets**

  In some cases, the modeler may wish to see a subset of objects defined by the metamodel. *Sets* are used to define a subset of objects that may be viewable at some point in time to the user when creating a model in the domain specific environment. This limits the objects from which the user can choose to build a model. This component is useful in situations where the domain expert wishes to view only objects pertaining to a particular state of a model under construction.

- **Aspects, Attributes, and Preferences**

  An *aspect* is a group of parts that exist or are visible in a particular view. This is useful for large or complex paradigms where too many parts are displayed at once for a given level of hierarchy. Aspects are determined by the modeling paradigm.

  All FCOs can have *attributes*. An attribute is a property of an object that is expressed textually as an annotation to the object. The possible text includes numbers, constants, or a finite set of symbolic characters. Typically objects have multiple attributes which are annotated to the object through entry fields, menus, or buttons. The modeling paradigm defines the attributes that are associated with each object in the metamodel. An interpretation of these attributes is left for the model interpreters and constraint checker.

  Preferences are paradigm-independent properties of objects that are defined by the GME itself. Each FCO has a different set of preferences. Some preferences include text color, line type, etc. They are inherited from the paradigm definition through type inheritance.

Interpreters

Model translators called model interpreters are designed to work with all models created using the domain-specific environment for which they are designed. Implementing model interpreters in GME is typically done after the domain expert has verified that valid models can be constructed in the domain. An interpreter is a library of interfaces and classes that are specific to the modeling paradigm. The designer of the domain usually annotates the pre-constructed interfaces and classes to give any model constructed in the domain a semantics. Interpreters translate models into physical artifacts, like a text file or another object.

Constraint Manager

GME contains a Constraint Manager, which is compliant with the OCL specification, to check models before they are interpreted. It is up to the designer of the domain to decide what should be checked. Without going into too much detail about the Constraint Manager, it simply limits the user to the models that are built in the domain. The Constraint Manager can be used to check whether or not a connection between two parts are valid, for example. The Constraint Manager is event-driven in that it provides immediate feedback to the user if a violation of rules or semantics of the modeling paradigm is encountered. The designer of the domain annotates the UML diagrams in the metamodel with OCL constraints that models the semantics or rules of the domain.

4.4.2 FTDF Domain Construction

As one may recall from Section 4.4.1, one of the first and most important steps to domain modeling in GME is to specify the paradigm. The paradigm is used to define the finite set of visual objects used in the FTDF domain. The objects contain attributes [2] that are defined within the FTDF paradigm and are characteristic of FTDF models. An instantiation of the FTDF paradigm allows the designer to use elements of that set of objects to construct an application model. The primary visual objects that may be used in the GME graphical interface are *Ports*, *Channels*, *Actors*, and *ECUs*. Note that ECUs are not a part of FTDF semantics, however, an ECU becomes an important object when one considers a *mapped* FTDF model which has a notion of architecture. Figure 4.3 gives the UML class diagram for the FTDF paradigm.

Ports

A port is a typed, atomic object in the FTDF paradigm. It can be of type *input* or *output*. Furthermore, input and output ports can be either *internal* or *external*. Figure 4.3 depicts a UML class diagram of the FTDF paradigm. Input and output port types are used to constrain the designer to creating connections between input/output ports and not input/input or output/output connections. Internal and external types are used by the Interpreter to identify whether or not a port is connected to actors on the same ECU or on a remote ECU. Ports contain two attributes, one that allows the designer to specify the data type for the port and another which specifies the number of buffers the port should contain.

[2]Attributes here refers to GME attributes as explained in Section 4.4.1.

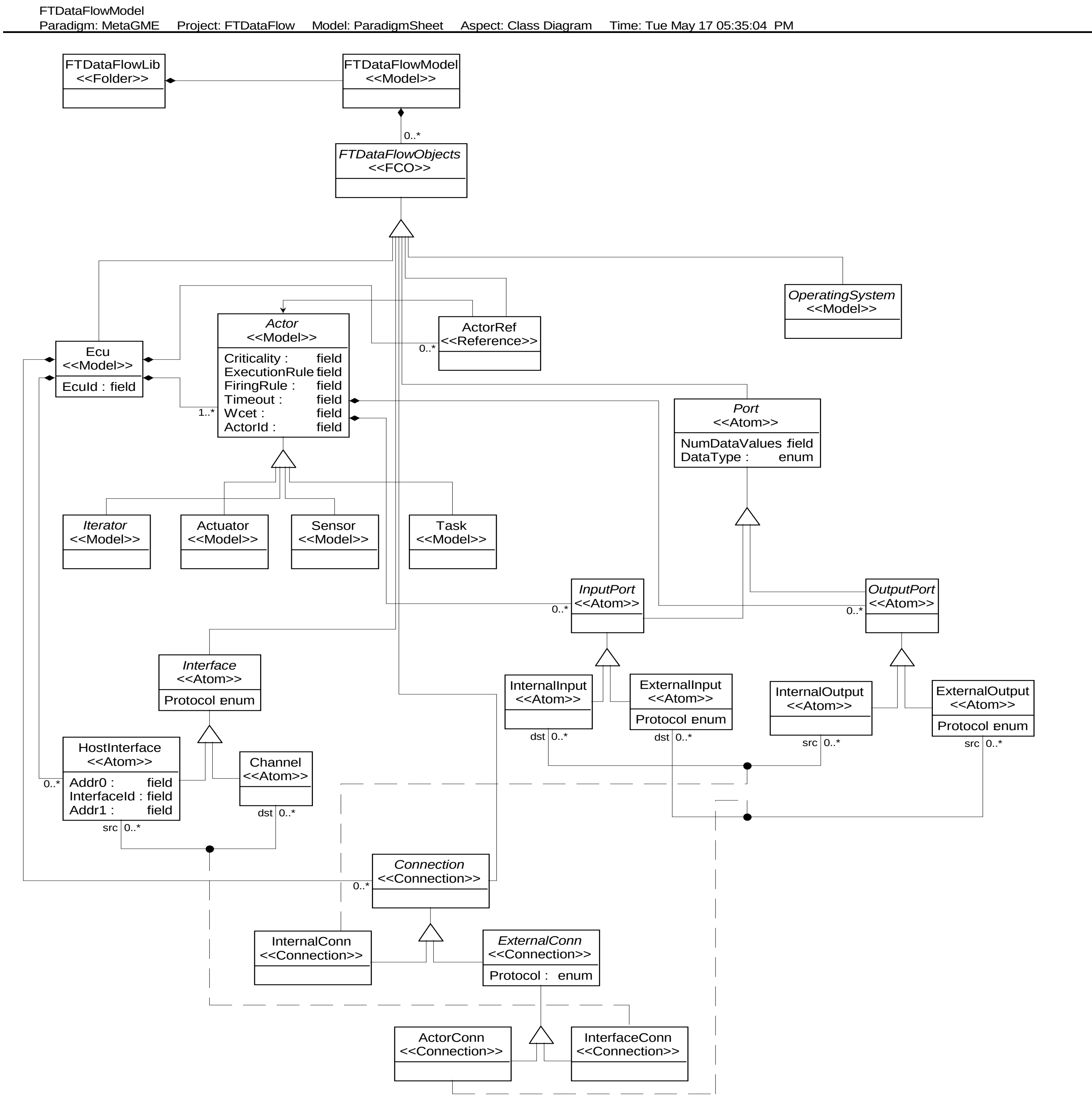


Figure 4.3. A UML diagram of the FTDF paradigm.

Channels

A channel is a typed object that forms a connection between two ports. As mentioned in the above section, connections can only occur between input and output ports. This helps to distinguish between a source actor (an actor producing a token) and target actor (actor receiving an output token).

Actors

Actors are the central objects in this domain. They are typed and hierarchical. An actor must contain at least one port, depending on the type of actor. GME eases the use of adding, deleting, or modifying actor types in the domain by modifying the UML diagram structure in the paradigm definition. Attributes of an actor type includes **Criticality**, **ExecutionRule**, **FiringRule**, **Timeout**, **Wcet**, and **ActorId**. The attributes give the designer enough flexibility to specify parameters that may be used to configure the runtime platform. The **ExecutionRule** and **FiringRule** are fields that allows the designer to specify code that will be used for the firing function and firing rule respectively. The code should be implemented as it would be implemented in the runtime platform by hand.

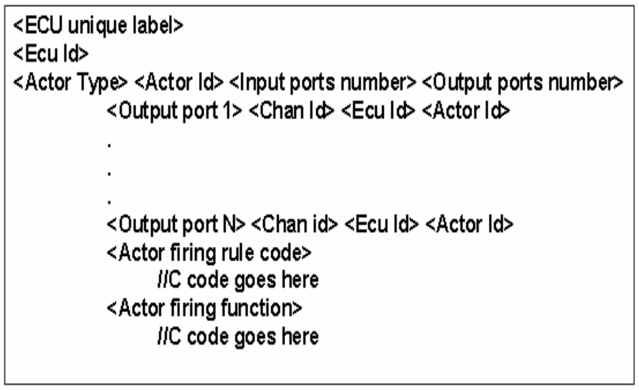
ECUs

An ECU object is a hierarchical abstraction of a processing element. It contains at least one actor object. Structural relationships between different ECUs are captured with port and channel objects. ECUs act as containers to hold actor instances. Communication between actors on remote ECUs occur directly between actors using reference objects. Future implementation may change that syntax to better reflect the FTDF communication media objects.

4.4.3 Interpreter Design

GME has a C++ based architecture that is used to build interpreters (model translators) called the Builder Object Network version 2 (BON2) component. When an interpreter is created, a set of interfaces for objects in a paradigm are produced that allow the designer to implement customizable C++ code. As an example, the FTDF paradigm has an actor object. The interface produced by the BON2 component is shown below.

```
bool FTDataFlowVisitor::visitActor( const Actor& object )
```



```
<ECU unique label>
<Ecu Id>
<Actor Type> <Actor Id> <Input ports number> <Output ports number>
            <Output port 1> <Chan Id> <Ecu Id> <Actor Id>
            .
            .
            .
            <Output port N> <Chan id> <Ecu Id> <Actor Id>
            <Actor firing rule code>
                  //C code goes here
            <Actor firing function>
                  //C code goes here
```

Figure 4.4. A template for the configuration file generated by the FTDF interpreter.

```
{
  // Code for manipulating Actor objects in the interpreter
  // will go here.
  return true;
}
```

Within the above function definition, the designer places customized code that tells the Interpreter what to do with every actor object encountered in the FTDF application model when the Interpreter is invoked. The FTDF interpreter utilizes these interfaces generated by BON2 to traverse FTDF objects. If an object has attributes, then the attributes are also accessible to the Interpreter designer. Structural model checking in GME can be done using the interpreter, the constraint manager, or by carefully defining structural relationships amongst paradigm objects using the UML class structure. For the FTDF paradigm, the Interpreter design and paradigm definition checks an application model for proper construction by checking the requirements for a legal FTDF graph. The FTDF Interpreter also serves the purpose of creating a configuration file of the FTDF application for which the Interpreter is invoked.

An example of the configuration file that is produced by the Interpreter is in Figure 4.4. The configuration file contains information about each ECU and actor in the FTDF application model. It contains the topology of the FTDF application model, including the type of actors, number of ports for each actor, a unique actor identification number, and code for the firing function and firing rule. The configuration can be parsed to complete the configuration of the runtime platform.

Chapter 5

Fault Tree Analysis

Designing cost-sensitive real-time control systems for safety-critical applications requires a careful analysis of both performance versus cost aspects and fault coverage of fault-tolerant solutions. This further complicates the difficult task of deploying the embedded software that implements the control algorithms on a possibly distributed execution platform (as it is typical, for instance, for automotive applications). Fault trees are commonly used models for obtaining dependability metrics of safety-critical systems. The designer first constructs a fault tree model of the system under analysis. Once a fault tree is constructed, a number of existing commercial analysis tools to assess dependability metrics of the system can be used. In this chapter, a synthesis technique for building a fault tree that models how component faults may lead to system failure is introduced. This synthesis technique and its analysis capability are added to the existing synthesis-based design methodology.

5.1 Background

Fault tree analysis (FTA) is a top-down approach to failure analysis: it starts with an undesirable event called a *top-level event*, then determines all the ways the event may occur in the system. Before proceeding, the terminology of faults and failures is briefly discussed. A *fault* is an undesirable event. In this context, an event is the state of a system or a system component that may be induced by the presence or absence of a proper command, or it may be caused by a failure. A *failure* is a loss of functionality by a system or system component. All failures cause faults, but not all faults are caused by failures. To reliability and safety engineers, this distinction is very important.

FTA can be used to address questions such as, "does the chosen design meet reliability and safety requirements?" and "what are the most critical components in the chosen system?" This allows a number of design options to be judged reasonably. FTA was initially developed in 1962 by Bell Telephone Laboratories for use with the Minuteman system for the United States Air Force [18]. The technique was later adopted and applied by the Boeing Company as one its logic analytical techniques. FTA is only one of many analytical tools and techniques to assess system safety. Some other techniques include Failure Mode and Effects Analysis (FMEA) and Event Tree Analysis (ETA), amongst many others. FMEA determines and evaluates effects of sub-component failures on a system, and ETA organizes, characterizes, and quantifies potential accidents. These techniques are accepted and applied to numerous safety-critical systems in the domain of reliability and safety engineering.

FTA uses a fault tree, such as the one in Figure 5.1 as the data structure to represent the relationship between potential hazards and their influence on system reliability. The top-level event of a fault tree is an abstraction of a system failure or malfunction. In FTA, the top-level event represented by the root of the tree. *Static* fault trees uses combinatorial relationships that can be expressed by Boolean functions. *Dynamic* fault trees, on the other hand, model failures that may not be expressed by combinatorial logic, like the use of Markov Chains to retain history of component failures. Fault trees have been used extensively to perform various kinds of dependability analyses [29]. For example, fault trees can be used to derive the system failure rate, or the sets of faults necessary to achieve a system failure (i.e. cut sets). Fault trees also indicate the system sensitivity to faults and to their failure rates and failure modes. Once a fault tree is generated, existing commercial tools can be used to perform these types of analyses automatically.

Although fault trees enable powerful analysis, they are manually constructed by experienced reliability and safety engineers. For complex hardware software systems, this process is very tedious and error-prone. The construction, in the general case, requires a very detailed knowledge of the system being modeled, including an understanding of the fault model used and how faults and failures are identified. However, since our approach is based on FTDF, this task can be made formal. In fact, FTDF specifies how faults propagate and how they affect the execution of the controller. This formal infrastructure allows the use of a synthesis-based technique for generating the fault tree automatically.

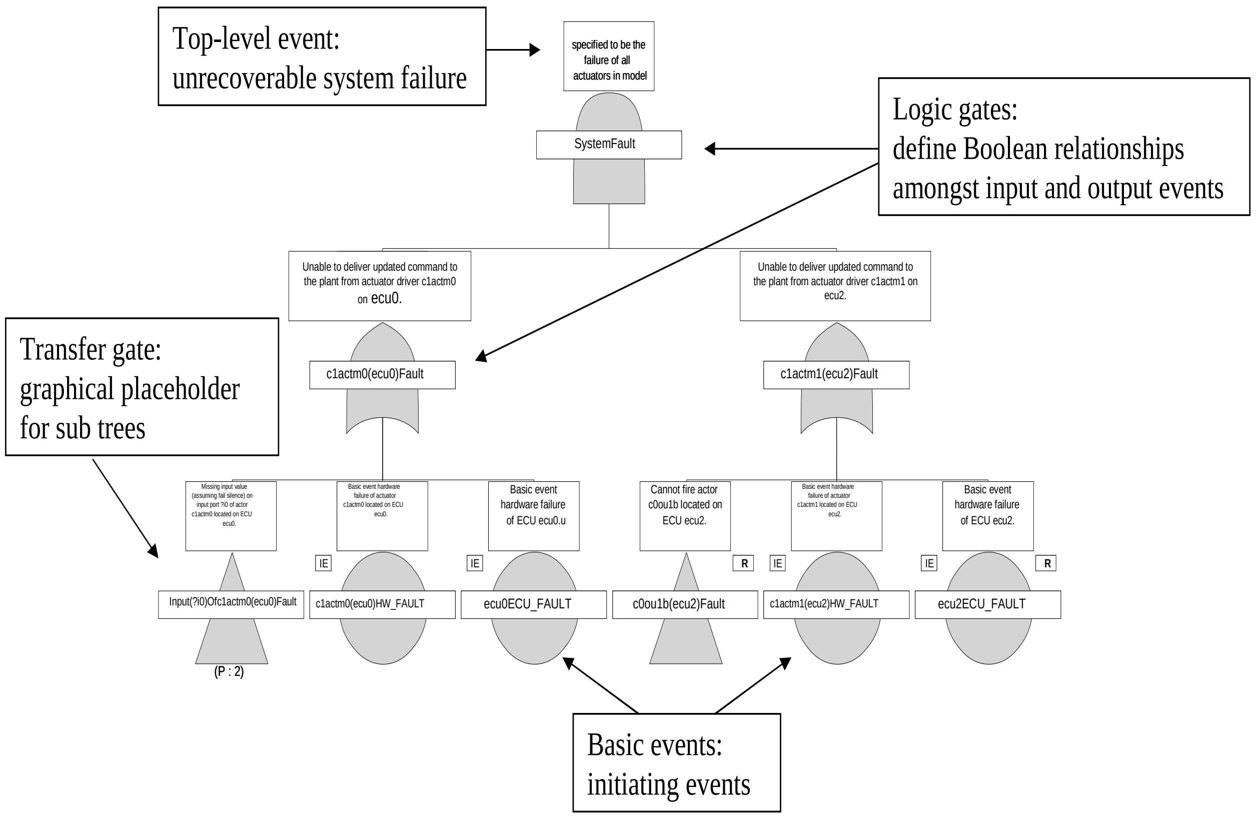

Top-level event:
unrecoverable system failure
specified to be the failure of all actuators in model
SystemFault
Logic gates:
define Boolean relationships amongst input and output events
Unable to deliver updated command to the plant from actuator driver c1actm0 on ecu0.
Unable to deliver updated command to the plant from actuator driver c1actm1 on ecu2.
Transfer gate:
graphical placeholder for sub trees
c1actm0(ecu0)Fault
c1actm1(ecu2)Fault
Missing input value (assuming fail silence) on input port ?i0 of actor c1actm0 located on ECU ecu0.
Basic event hardware failure of actuator c1actm0 located on ECU ecu0.
Basic event hardware failure of ECU ecu0.u
Cannot fire actor c0ou1b located on ECU ecu2.
Basic event hardware failure of actuator c1actm1 located on ECU ecu2.
Basic event hardware failure of ECU ecu2.
IE
IE
R
IE
IE
R
Input(?i0)Ofc1actm0(ecu0)Fault
c1actm0(ecu0)HW_FAULT
ecu0ECU_FAULT
c0ou1b(ecu2)Fault
c1actm1(ecu2)HW_FAULT
ecu2ECU_FAULT
(P : 2)
Basic events:
initiating events

Figure 5.1. A sample fault tree generated by the Item Toolkit.

5.2 Related Work

There has been prior work in the area of fault tree synthesis and design flows for safety critical systems such as [31] [10] [26] [12]. The work most closely related to the work presented in this paper is [31], where a methodology is presented that enables fault tree synthesis based on a composition of both a hazard analysis[1] on the architectural components and a functional fault tree annotated with the functional components[2]. Having the composition of the two enables the system level fault tree to be synthesized. The basic fault events of the architectural components are assumed to be either output omission, timing failures or value domain failures.

This work differs in the sense that there is a formal model of computation of the functionality (the FTDF MoC) that describes how a function can fail, how the architectural fault modes[3] are manifested in the functionality and also how the faults of the mapped components are composed together. The fault tree also comprehends how architectural failures are propagated through the functional data dependencies. Since there is a formal definition of how the functionality behaves when mapped onto a shared architecture, *common mode failures* are easily reflected in the fault tree as multiple events of the same type in different fault tree branches. A common mode failure describes the failure of two or more components in a system or sub-system due to a single specific event.

5.3 Fault Tree Synthesis

In this section, an algorithm for automatic fault tree generation from a mapped FTDF graph is formulated and described. The static fault tree generated is represented in a Boolean format. The fault tree will describe how faults in the execution platform may lead to faults in the functionality and ultimately to violations of the specifications, i.e. to system failures. A recursive algorithm operates on the deployed FTDF graph model to produce a system fault tree. The problem is defined more precisely below.

[1]Hazard analysis is a systematic method of finding out the manner in which an architectural component fails.
[2]Component here is defined as a functional component that has a specific input and output interface.
[3]In the current implementation, it is assumed that the architecture can only fail silently.

5.3.1 Problem Formulation

A redundant mapped FTDF graph $\mathcal{L}_{FT}$, is generated by the synthesis algorithm described in [21], and it is given as

$$\mathcal{L}_{FT} = (L_{V_{FT}}, L_{E_{FT}}), \tag{5.1}$$

where $L_{V_{FT}} = (P \cup C) \times V$ is the set of vertices and $L_{E_{FT}}$ is the set of edges. In $L_{V_{FT}}$, P is the set of ECUs, C is the set of channels, and V is the set of actors and media, as defined in Section 3.4. A vertex $l \in L_{V_{FT}}$ with $l = (r, v)$ means that an actor or medium v is mapped to resource r. An edge $e \in L_{E_{FT}}$ with $e = (l_1, l_2)$, $l_1 = (r_1, v_1)$, and $l_2 = (r_2, v_2)$ connects l_1 to l_2. To illustrate, consider two possible cases of mapping.

- *Two actors* are mapped on the same ECU, the first actor delivers data to the second and no channel is involved, i.e. $p_1 \in P$ and $a_1, a_2 \in A$, then $l_1 = (p_1, a_1)$ and $l_2 = (p_1, a_2)$.
- *One actor* is mapped on an ECU, it transmits data on a channel, i.e. $p_1 \in P, c_1 \in C, a_1 \in A, m_1 \in M$, then $l_1 = (p_1, a_1)$ and $l_2 = (c_1, m_1)$.

Graph $\mathcal{L}_{FT}$ preserves the structural dependencies between its components as specified in the unmapped FTDF graph, $\mathcal{G}$.

A fault tree is a representation of Boolean relationships among fault events. Fault events e are Boolean variables which take values in $B = \{0, 1\}$, when an event takes the value 1, i.e. $e = 1$, the corresponding fault has occurred, conversely, when $e = 0$ the corresponding event has not occurred. The set of *basic* events I_B designates the input events to a fault tree. In our case, basic events corresponds to faults in system components, such as electronic circuits or communication channels. A fault tree contains a single output event, or root event, called a *top-level* event denoted by e_T. The top-level event indicates the failure of the system.

The fault tree describes how the faults of basic components (basic faults) may lead to the system failure. A fault tree is composed of a set of *gates* F_G. A gate $v_g \in F_G$ has an associated Boolean symbol g, and has one output event e_g. With a slight abuse of notation, we will say that an event $e_g \in F_G$, if it is the output event of a (unique) gate $v_g \in F_G$. The set $I_{v_g} \subseteq F_G \cup I_B$ denotes the input events to gate v_g. An input event $e_1 \in I_{v_g}$ is either the output event of some gates, i.e. $e_1 \in F_G$, or one of the basic events, i.e. $e_1 \in I_B$. Each gate v_g is assigned a Boolean function f_{v_g} that computes the value of the output event e_g given the values of the input events. A function f_{v_g} is defined as

$$f_{v_g} : B^N \to B \tag{5.2}$$

where $N = |I_{v_g}|$ is the number of inputs to gate v_g. The function f_{v_g} of gates in a fault tree has a one-to-one correspondence between the Boolean logic representation and the fault tree representation [29].

A fault tree $\mathcal{F}$ can be formulated as a directed, acyclic graph

$$\mathcal{F} = (F_G, I_B, F_N), \tag{5.3}$$

where F_G is the set of vertices, I_B is the set of basic events, and $F_N \subseteq (F_G \cup I_B) \times F_G$ is the set of directed edges connecting the vertices. An edge $n \in F_N$ is a pair of vertices $n = (v_s, v_d)$ such that $v_s \in F_G \cup I_B$ is the source vertex and $v_d \in F_G$ is the sink vertex.

Problem Statement: *Given a fault tolerant graph $\mathcal{L}_{FT}$, the top-level event e_T (and the gate which outputs it), generate a fault tree $\mathcal{F} = (F_G, I_B, F_N)$ and a correspondence map $f_{\mathcal{F}} : L_{V_{FT}} \rightarrow F_G$, such that:*

- *the set of basic events I_B is in bijection with $P \cup C \cup A_S \cup A_{Act}$ and indicates the failure of resources in the architecture*
- *$f_{\mathcal{F}}(l)$, where $l = (p, a)$, returns the gate $v_l \in F_G$ that indicates the faulty/missed execution of actor a on ECU p*
- *$f_{\mathcal{F}}(l)$, where $l = (c, m)$, returns the gate $v_l \in F_G$ that indicates the faulty/missed transmission of the data-dependency m on channel c*

It should be noted that the top-level event e_T can be derived by the semantics of the FTDF model of computation. For example, each output actor is annotated by the designer with the minimum number of actuators that it must be able to update in order to achieve a correct actuation of the control algorithm. In the pendulum example this number is 1, i.e. at least one of the two actuators should be updated. Correspondingly, e_T could be described as the output of an *AND* gate where the inputs are the failure to update the two actuators $ACT0$ and $ACT1$.

In general, designers may want to specify different top-level events to assess other aspects of the system response to faults. For this reason, we take the top-level event as an input to the fault-tree synthesis problem.

5.3.2 Fault Tree Generation Algorithm

The fault tolerant deployment graph, $\mathcal{L}_{FT}$, exhibits strong structural dependencies amongst actors in this graph. This dependency provides the necessary information to build the fault tree of a system modeled using FTDF. Thus, a recursive procedure is implemented in algorithm *GenerateFaultTree* to traverse the fault tolerant graph and generate a fault tree of the system.

The algorithm begins by creating and adding a user-defined top-level event, e_T to the fault tree $\mathcal{F}$. The designer provides the top-level event as a function of the actuator fault events. Algorithm *GenerateFaultTree* proceeds with generating subtrees for each actuator $a_{act} \in A_{Act}$ in $\mathcal{L}_{FT}$ using the recursive operation, *DevelopSubTree*. A fault in an ECU, channel, actuator or sensor HW is a basic event. The recursion ends when a sensor actor $a_s \in A_S$ is encountered in the fault tolerant deployment graph. The subtrees of each actuator are combined in a gate, as specified by the designer, to form event e_T.

Algorithm *[GenerateFaultTree]:*

Input: $\mathcal{L}_{FT}, e_T$
Output: $\mathcal{F}$

```
**GenerateFaultTree** ($\mathcal{L}_{FT}, e_T$) {
    For $a_i \in A_{ACT}$,
        Let $p_i \in P$ be s.t. $l_{a_i} = (p_i, a_i) \in L_{V_{FT}}$,
        $e_i = \textbf{DevelopSubTree}(l_{a_i})$;
    End For
    $e_T = \text{AddGate}(e_T, f_{e_T,\forall e_i}(e_i))$;
**End GenerateFaultTree**
```

Algorithm *DevelopSubTree* is the core of the fault tree synthesis tool which performs the recursion. When a vertex $l \in L_{V_{FT}}$ of graph $\mathcal{L}_{FT}$ is visited, the algorithm creates and stores a subtree $\mathcal{F}_{Sub_l}$ at l. This subtree is then appended to $\mathcal{F}$, and the operation is called recursively on each input of vertex l until a sensor, $a_S \in A_S$, is reached. When a sensor is reached, the recursion ends.

The algorithm *DevelopSubTree* first generates initial parameters. The operation *pre(a)* returns the set of sources (communication media) of actor a, and *minFire(a)* returns the number of inputs that are needed to fire actor a, and it depends on the firing rule of a. The number of inputs to fire an actor is given by the designer as a parameter when the FTDF graph is constructed. Next, a root event of subtree $\mathcal{F}_{Sub_l}$ is created and a number of events based on the type of actor encountered are also created. For example, event e_3 is created as a sensor basic event if $a \in A_S$ or an actuator basic event if $a \in A_{Act}$. A sensor or actuator basic event corresponds to a fault in the electro-mechanical HW of the sensor or actuator. The algorithm adds an *OR* gate with events e_1, e_2, and e_3 (if applicable). Here, e_1 corresponds to a fault of the ECU, e_2 is a fault that describes when the actor cannot fire because of missing tokens. If $a \in A_S$, the algorithm terminates the recursion immediately.

For each medium that connects actors a and a_j, an input fault event is created for that medium. The input fault event specifies that a medium was unable to deliver a token at the input of actor a.

The algorithm then checks for the location of the medium, whether it is a connection on the same ECU via memory or between actors or if its a channel that connects two actors across different ECUs. If the medium is on the same ECU, no immediate event is created and the algorithm is called recursively to further develop that event. The algorithm then creates events for each edge l_k, and it adds each event to an *OR* gate, to model the fact that remote data from actor instance l_k, is not delivered to actor instance $l = (p, a)$ if either the channel used is faulty or the remote actor fails to execute. The input edges to l_k are further developed by a recursive call to *DevelopSubTree*. At the end of the algorithm, notice that the second level of the tree (second from the root event) is created with a *VOTE* gate. The *VOTE* gate is useful to create a Boolean relationship between a subset of inputs to itself. This is a one-to-one correspondence to the input and arbiter actors, which have partial firing rules. More specifically, the *VOTE* gate requires that "x-out-of-y" input events must occur before an output event to occur at that gate. It is intuitive to see that when $x = y$, this simplifies to an *AND* gate and when $x = 1$, this simplifies to an *OR* gate. As an example, let $y = pre(a \in A_{In})$ and $m = |minFire(a)|$. Then for the *VOTE* gate of actor a, $x = (y - m + 1)$. This means that if x tokens are not present at the input of actor a, then the *VOTE* gate for actor a produces a fault event $e = 1$ at its output. Essentially, the algorithm composes subtrees to create the system fault tree.

Algorithm *[DevelopSubTree]:*

Input: $l = (p, a) \in (P \times A) \subset L_{V_{FT}}$
Output: $\mathcal{F}_{Sub_l}$

```
DevelopSubTree ($l$) {
    Let $N = |pre(a)|$;
    Let $M = minFire(a)$;
    $\mathcal{F}_{Sub_l}$ = CreateActorEvent($l$);
    $e_1$= CreateEcuBasicEvent($p$);
    $e_2$= CreateInputFaultEvent($a$,$N$);
    If $a \in A_S$ Then $e_3$= CreateSensorBasicEvent($a$);
    If $a \in A_{Act}$ Then $e_3$= CreateActuatorBasicEvent($a$);
    $\mathcal{F}_{Sub_l}$ = AddGate($\mathcal{F}_{Sub_l}$, $OR(e_1, e_2, e_3)$);
    If $a \in A_S$ Then return $\mathcal{F}_{Sub_l}$; //terminal case, end recursion

    For $m_j \in pre(a)$,
        $a_j = pre(m_j)$;
        $e_j$= CreateInputFaultEvent($m_j, l$);
        If ($l_{local} = (p, a_j) \in pre(l)$) Then
            $e_{local}$ = DevelopSubTree($l_{local}$);
        For $l_k = (c, m_j) \in pre(l) \cap (C \times \{m_j\})$,
            $e_k$= CreateRemoteInputEvent($l_k$);
            $e_c$ = CreateChannelBasicEvent($c$);
            $e_{ra}$ = CreateRemoteActorsEvent($l_k$);
            $e_k$ = AddGate($e_k, OR(e_c, e_{ra})$);
            For $l_r \in pre(l_k)$,
                $e_r$ = **DevelopSubTree($l_r$)**;
            End For;
            $e_{ra}$ = AddGate($e_{ra}, AND_{\forall e_r}(e_r)$);
        End For
        $e_j$ = AddGate($e_j, AND_{\forall e_k}(e_k, e_{local})$);
    End For
    $e_2$ =AddGate($e_2$, $VOTE_{\forall e_j}(N, M, (e_j))$);
    return $\mathcal{F}_{Sub_l}$; } **End DevelopSubTree**
```

Chapter 6

Results

6.1 Software Synthesis: Simple Example

A simple example is given to illustrate the code generation capability of the FTDF design environment. The example application is illustrated in Figure 6.1. Figure 6.2 illustrates the application modeled in the GME graphical user interface using FTDF objects.

The example given is simple, and it contains only five actors. The purpose was to have the ability to compare to a manual implementation. The actors are a Source and Sink actor denoted as “Sensor” and “Actuator” respectively, two Regular actors denoted as “Controller” and “Fine Controller” in Figure 6.1, and an Input actor denoted as “Arbiter”. The firing function and firing rules of each actor in the example is simple; each actor contains less than 20 lines of code for both the firing rule and firing function each. Figure 6.3 gives a summary of the estimated time and effort to create the FTDF design environment. The runtime platform is implemented in 1084 lines of C code. Construction of the FTDF paradigm in GME, the interpreter design, and the runtime platform are created once and can be reused for subsequent applications. To implement the FTDF

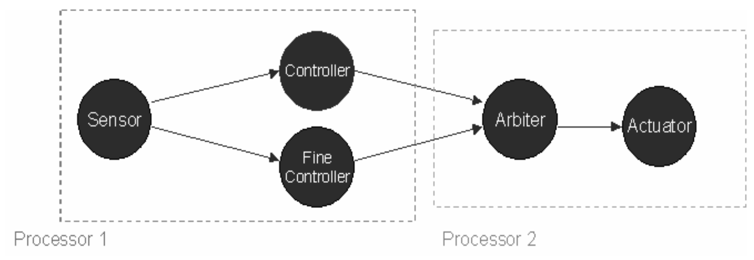
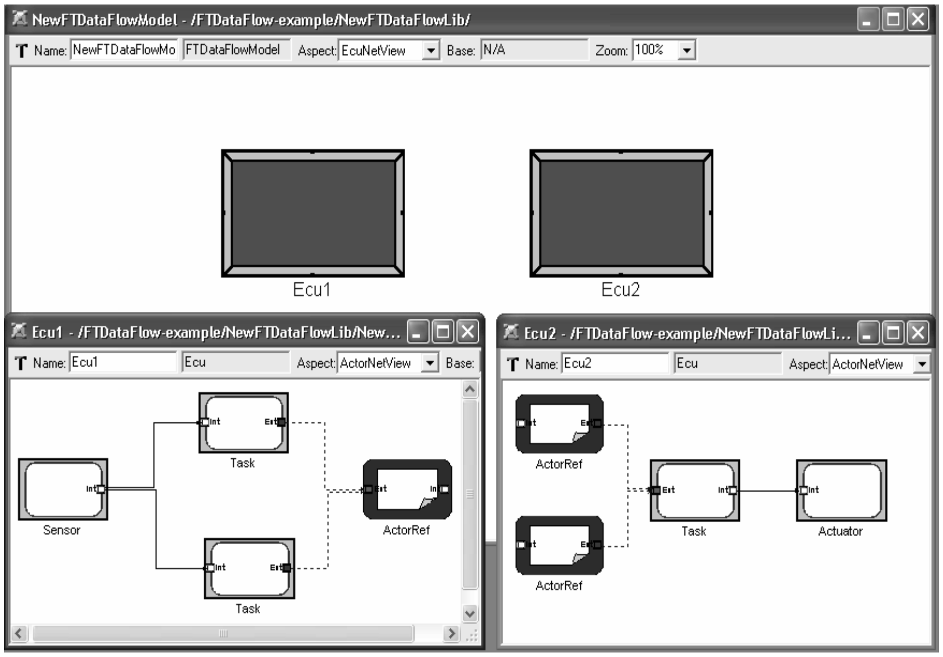


Figure 6.1. A simple example.



NewFTDataFlowModel - /FTDataFlow-example/NewFTDataFlowLib/

Name: NewFTDataFlowMo FTDataFlowModel Aspect: EcuNetView Base: N/A Zoom: 100%

Ecu1 - /FTDataFlow-example/NewFTDataFlowLib/New...

Name: Ecu1 Ecu Aspect: ActorNetView Base:

Ecu2 - /FTDataFlow-example/NewFTDataFlowLi...

Name: Ecu2 Ecu Aspect: ActorNetView

Figure 6.2. FTDF modeling of the simple example in GME.

Task	Time (hours)
FTDF Paradigm Development	12
Interpreter Design	40
Runtime Environment	85

Figure 6.3. Implementation time for key components of software synthesis flow.

	Time to enter application (hours)	Time to produce configuration (Minutes)
GME	1	0.15
Manual	2	20

Figure 6.4. Time for entering example application and configuring the runtime platform.

interpreter in GME, over 800 lines of code were added to the BON2 interfaces. Figure 6.4 contrasts the amount of time it takes to configure the runtime platform manually versus the automatic method prescribed by the FTDF interpreter. The results presented in Figure 6.4 suggests the gain of using the FTDF paradigm in GME to specify and configure an application for the runtime platform can scale with the application size. An application based on the configuration file for the simple example is given in the code snippet below.

```
/* Main program for testing.
*  ecu.c
*  Last updated:  11/13/03
*  Version: 5.1
*
*  Notes
*
*  - this file symbolizes user code on a single processor.
*
*/


/***********************************************************
Headers
***********************************************************/


#include "ukernel.h"


/***********************************************************
Declare User Global Variables and Data Structures
***********************************************************/


//sample data, simulating a sensor's readings
dtype sensData = 0;

//declaring actors and ports
actor a0, a1, a2, a3, a4, a5;
actorPort a0pt, a1pt, a2pt, a3pt, a4pt, a5pt;
actor *actList[5] = {&a0,&a1,&a2,&a3,&a4};

//declaring this ecu node by:
//its globally unique id and address info
int ecuid = 0;
nodeAddr host = {0, "127.0.0.1", 4040};

int interfid[1] = {-1}; //interface id, it is only declared and initialized here
```

```
//Destination lists (specifies output buf connections)
//for each actor's output, where each ouput could have multiple
//dest lists.
//Order: src buf index, outgoing channel from host ecu (which can be obtained
// by calling "setInterface" and using its return value. Also, use "-1"
// when not connecting any channel interfaces),
// dest ecu id (unique identifier for each ecu),
// dest actor index, dest actor buffer
//

//Below are UNIQUE destination lists (2-d arrays).
int dlist_a00[1][5] = {0,-1,0,1,0};
int dlist_a10[1][5] = {0,-1,0,2,0};
int dlist_a11[1][5] = {1,-1,0,3,0};
int dlist_a20[1][5] = {0,-1,0,4,0};
int dlist_a30[1][5] = {0,-1,0,4,1};
int dlist_a40[1][5] = {0,-1,0,0,0};

//if an actor has multiple outputs, uKernel expects
//multiple dest lists, one dest list per output.
int *dlists_array_a0[1] = {&dlist_a00[0][0]};
int *dlists_array_a1[2] = {&dlist_a10[0][0], &dlist_a11[0][0]};
int *dlists_array_a2[1] = {&dlist_a20[0][0]};
int *dlists_array_a3[1] = {&dlist_a30[0][0]};
int *dlists_array_a4[1] = {&dlist_a40[0][0]};

//declare input and output token buffers
dtype dRoom[8]; //user allocated space for tokens for
//each of the 4 actors input and output
//buffers
token a0iBuf[1] = {-1,FALSE,0,NULL};
token a0oBuf[1] = {-1,FALSE,0,NULL};

token a1iBuf[1] = {-1,FALSE,0,NULL};
token a1oBuf[2] = { {-1,FALSE,sizeof(dtype),dRoom + 0},
{-1,FALSE,sizeof(dtype),dRoom + 1} };

token a2iBuf[1] = {-1,FALSE,sizeof(dtype),dRoom + 2};
token a2oBuf[1] = {-1,FALSE,sizeof(dtype),dRoom + 3};

token a3iBuf[1] = {-1,FALSE,sizeof(dtype),dRoom + 4};
token a3oBuf[1] = {-1,FALSE,sizeof(dtype),dRoom + 5};

token a4iBuf[2] = { {-1,FALSE,sizeof(dtype),dRoom + 6},
{-1,FALSE,sizeof(dtype),dRoom + 7}};
token a4oBuf[1] = {-1,FALSE,0,NULL};
```

```
/*********************************************************
Create User Actor Execution/Firing Rules
*********************************************************/


void a1Exec (actor *a) {

//Sensor execution function (sensor driver).
//Data originates from this actor in this application,
//on every iteration
//and is sent to FTDF network.  It expects
//only one input token in its input buffer which is
//a trigger from the iterator.  This actor uses
//the "and" firing rule as defined in the API to await
//on a trigger from the iterator. When writing outputs,
//it sets the outgoing tokens with new data, tags
        //output tokens with latest epoch and validity.
        //User should NOT alter dsz or data (ptr) fields in
        //actor struct, because those fields are used
//to record the amount of memory that has been
//allocated to store tokens.
//NOTE:  User can access actor port information.
//All actors should do the following:
// 1. read tokens and/or triggers from input buf
// 2. perform some execution
// 3. write tokens to output buf
//Arguments:
//aPort, ptr to actorPort struct
        //NOTE:  For this actor, since its input is a trigger
        //token from the iterator, it does not need to read
        //any data from input token buf, nor does it need to
        //read the valid field because the valid is assumed
        //to be TRUE coming from the iterator since the
        //"and" firing rule will only fire if all of its
        //input tokens are valid and current.

  token temp;
  dtype sdata[1];

  //compute something
  sdata[0] = sensData++;

  //produce a temp token (read epoch from input token)
  temp = readToken(a->port->inBuf+0);
  temp.valid = TRUE;
  temp.data = sdata;
  temp.dsz = 1 * sizeof(dtype);
  //write to output buf - expects 2 output tokens
  writeToken (&temp, a->port->outBuf+0);
  writeToken (&temp, a->port->outBuf+1);
}

void a2Exec (actor *a) {

//Actor 2 uses the "and" firing rule.
//It multiplies the input data by 2 then adds
//2 to the product.

  token temp;

  //read input - expects 1 input token
  temp = readToken(a->port->inBuf+0);

  //compute
  temp.data[0] = (temp.data[0] * 2) + 2;
  temp.valid = TRUE;

  //write to output - expects 1 output token
  writeToken(&temp,a->port->outBuf+0);
}

void a3Exec (actor *a) {

//Actor 3 uses the "and" firing rule.  It takes
//input and multiply by 2, then adds 2 to the
//product.

  token temp;

  //read input - expects 1 input token
  temp = readToken (a->port->inBuf+0);

  //compute
  temp.valid = TRUE;
  temp.dsz = 1 * sizeof(dtype);
  temp.data[0] = (temp.data[0] * 2) + 2;

  //write to output - expects 1 output token
  writeToken (&temp,a->port->outBuf+0);
}

bool a4Fire (actor *a) {

//Firing rule for Arbiter actor.  It fires
```

```
//once it receives at least one valid input
//on time with an epoch that is greater
//than the current local actor epoch.
//NOTE: actorCode() in the API updates the local
//actor epoch to the max epoch received in input
//buffers. It does the update AFTER the actor fires,
//but BEFORE the actor's execution function.

  int i = 0;
  int validCt = 0; //valid tokens in buffer count
  token temp;

  for (i = 0; i < 2; i++) {  //buf with 2 inputs
    temp = readToken(a->port->inBuf+i);
    if ((temp.epoch > a->epoch) &&
      (temp.valid == TRUE)) {
      validCt++;
    } else {
      //clear token in buf and wait for
      //a valid token to come
      resetToken(a->port->inBuf+i);
    }
  }
  if (validCt >= 1) return TRUE;
  return FALSE;
}

void a4Exec (actor *a) {

  token temp;
  int i = 0;

  //read input - expects at least 1 valid input token
  do {
    temp = readToken(a->port->inBuf+i);
    i++;
  } while (temp.valid != TRUE);

  //show me the data
  if (temp.epoch > 0) {
    printf ("\n\tEpoch: %d\n", temp.epoch);
    printf ("\tValid: %d\n", temp.valid);
    printf ("\tData: %d\n\n", temp.data[0]);
    sleep(1);
  }

  //write to output - expects 1 output trigger
  //with updated epoch
  temp.epoch = a->epoch;
  temp.valid = TRUE;
  temp.dsz = 0;
  temp.data = NULL;
  writeToken (&temp,a->port->outBuf+0);
}


/***************************************************
Set Actors and Actor Ports - Initialize
****************************************************/


void userInit () {

//sets channel (physical network) interfaces.  Can be called multiple times and it
//automatically updates "interfNum" (transparent to user) as it is called.  Returns
//a channel id to reference the channel.  Note:  This is because the idea is such that
//a designer can specify multiple channels for each node (logical ECU).

//Args: pointer to a loading function for the channel, nodeAddr struct
//filled with host ecu's address info and global unique id, a list
//of nodeAddr structs that represent the nodes of the channel on the network
//excluding the host address info, and the number of nodes on that channel.
//NOTE: Must give a valid pointer to a loading function for the channel
//you wish to use, or NULL if this ecu is not connected to a channel (external communication li
//Use NULL for arg 3 if no channel is connected in order to route tokens
//on local ecu only.

interfid[0] = setInterface (NULL, &host, NULL, 0);

dlist_a00[0][1] = interfid;
dlist_a10[0][1] = interfid;
dlist_a11[0][1] = interfid;
dlist_a20[0][1] = interfid;
dlist_a30[0][1] = interfid;
dlist_a40[0][1] = interfid;

//set actor ports
setActorPort (&a0pt, 1, a0iBuf, 1, a0oBuf, dlists_array_a0);
setActorPort (&a1pt, 1, a1iBuf, 2, a1oBuf, dlists_array_a1);
setActorPort (&a2pt, 1, a2iBuf, 1, a2oBuf, dlists_array_a2);
setActorPort (&a3pt, 1, a3iBuf, 1, a3oBuf, dlists_array_a3);
setActorPort (&a4pt, 2, a4iBuf, 1, a4oBuf, dlists_array_a4);

//set actors
//Note: A0 is my iterator
setActor (actList[0], "ITERATOR", NULL, NULL, 0, &a0pt, 50, 50);
setActor (actList[1], "SENSOR", NULL, a1Exec, 0, &a1pt, 50, 50);
setActor (actList[2], "FUNCT0", NULL, a2Exec, 0, &a2pt, 50, 50);
setActor (actList[3], "FUNCT1", NULL, a3Exec, 0, &a3pt, 50, 50);
setActor (actList[4], "ABSTRACTOUT", a4Fire, a4Exec, 0, &a4pt, 50, 50);

//This function constrains the user to only creating one ECU per file/node.
setLocalEcu (ecuid,5,actList);
}


/*******************************************************
Main Process
*******************************************************/


int main(int argc, char **argv) {

  userInit (); //initialize actors and set to ukernel
  run(); //runs FTDF application
  return 0;
}
```

The code creates an application that will be executed by the runtime platform. In particular, the runtime platform will implement the application on a single PC using UDP over IP for network communications. Since the runtime platform is Linux/Unix compatible and it implements POSIX compatible threads, an application can map more than one process (i.e. ECU) to the same PC as a separate process. This is useful for simulation and prototyping convenience, when more hosts are needed than at hand given that the IP stack can handle both local and remote communication.

6.2 Fault Tree Analysis: Pendulum Example

In this section, the inverted pendulum example is used to illustrate some of the dependability metrics of fault trees generated by the fault tree synthesis tool. The controller is described by the FTDF graph in Figure 2.1. It consists of three position Sensors (SEN0, SEN1, SEN2) one Input actor (IN) that performs sensor integration and assesses the current pendulum position, two different controllers (FUNc, FUNf) that require different computing power, one Arbiter (AR) that selects the output of one of the controllers, and an Output (OUT) actor that directs the control action to two different Actuators (ACT0, ACT1). The execution platform is given in Figure 2.2 and consists of three ECUs (ECU0, ECU1, ECU2) and two communication channels (CH0, CH1).

Actor	**ECU0**	**ECU1**	**ECU2**
SEN0	X		
SEN1		X	
SEN2			X
IN		X	X
FUNc	X	X	
FUNf			X
ARB		X	X
OUT		X	X
ACT0	X		
ACT1			X

Table 6.1. Redundant mapping of the pendulum example.

Each ECU samples one of the three Sensors, while “ECU0” and “ECU2” drive Actuator “ACT0” and “ACT1” respectively.

The fault tolerance requirements used to drive the synthesis of the system deployment consist of:

- executing the entire algorithm in the absence of faults (default behavior) and
- in the presence of a single ECU fault, guarantee the execution of a subset of the FTDF graph, so that at least one of the Actuators is updated.

The critical set mandates the execution of at least one replica of

- the Input actor “IN”,
- the coarse controller “FUNc”,
- the Arbiter actor “ARB”,
- and the Output actor “OUT”.

In particular, in order to fire, actor “IN” requires that at least two of the Sensors deliver their data. The Arbiter actor can fire with one of its two inputs present. It is worth noting that the fine controller “FUNf” is not replicated, because it was not specified as part of the critical set.

The mapping given in Table 6.1 is guaranteed by construction to tolerate single ECU failures. However, there is no fine quantification of the degree of fault tolerance achieved by this particular implementation. For this, the fault tree synthesis tool is used, extracting a fault tree representation of the mapped system in Table 6.1. The analysis from the fault tree tells the designer how and why the system may fail.

The fault tree is then analyzed by the Item Toolkit [7], a commercial tool distributed by Item Software Incorporated. Among many analyses that can be performed, results from a cutset analysis, the dependability analysis, and the sensitivity analysis are presented to show the analysis metrics that may be used to judge a specific system mapping.

6.2.1 Cutset Analysis

The cutset analysis permits to identify the combinations of events that generate a system failure. The list of minimal cutsets for the mapped pendulum example is presented in Table 6.2. The cutsets are rows in the table. As an example, if an event occurs with SEN0 and SEN1 located in row 1 of the table, then a system failure occurs. Table 6.3 shows the channel mappings for the system. The first column of Table 6.3 features an actor in column 1 that is mapped to the ECU in column 2. The actor accepts inputs from the given channel in column 3 and produces outputs for the channel in column 4. The name of the basic events correspond to hardware faults in Sensors, Actuators, ECUs, or communication channels. As mentioned in Section 5.1, the basic events are determined by the designer since basic events may be as detailed as a power outage or as abstract as an actor failure. As expected, no single ECU failure leads to a system failure. Moreover, no single channel failure leads to a system failure, which is not guaranteed by the synthesis algorithm.

6.2.2 Reliability Analysis

The reliability analysis combines the reliability information about components into the reliability of the system. Starting from the mean time to failure (MTTF) and the mean time to repair (MTTR) of the basic events. Among other metrics, the system MTTF is computed.

In this simple example, a system lifetime is assumed to be 5000 *hours*. All basic events are assumed to have the same reliability:

$$MTTF(basic\ event) = 2000\ hours$$
$$MTTR(basic\ event) = 8\ hours.$$

The Item Toolkit returns,

$$MTTF(system) = 11015.81\ hours.$$

Based on the simple specification, one could expect a reliability higher than that of a single component failing ($MTTF(system) > 2000\ hours$). Having a fault tree now allows experimenting

Cutsets		
1	SEN0	SEN1
2	SEN0	SEN2
3	SEN1	SEN2
4	ECU0	ECU1
5	ECU0	ECU2
6	ECU1	ECU2
7	CH1	CH0
8	ACT0	ACT1
9	CH0	ECU1
10	CH0	ECU2
11	CH1	ECU0
12	CH1	ECU2
13	SEN0	CH1
14	SEN1	CH0
15	SEN2	CH0
16	SEN0	ECU1
17	SEN0	ECU2
18	SEN1	ECU2
19	SEN1	ECU0
20	SEN2	ECU0
21	SEN2	ECU1
22	ACT0	ECU2
23	ACT1	ECU0

Table 6.2. Cutsets for the pendulum example.

Actor	**ECU**	**Input**	**Output**
SEN0	ECU0		CH0
FUNc	ECU0	CH0	CH0
ACT0	ECU0	CH0,CH1	
SEN1	ECU1		CH1,MEM
IN	ECU1	CH0,CH1,MEM	CH0,MEM
FUNc	ECU1	MEM	CH0,MEM
ARB	ECU1	CH0,MEM	MEM
OUT	ECU1	MEM	CH1,MEM
SEN2	ECU2		CH1,MEM
IN	ECU2	CH0,CH1,MEM	CH0,MEM
FUNf	ECU2	MEM	CH0,MEM
ARB	ECU2	CH0,MEM	MEM
OUT	ECU2	MEM	CH0,MEM
ACT1	ECU2	MEM	

Table 6.3. Channel mapping of the pendulum example.

Event	**Fussell-Vesely**	**Birnbaum**	**Barlow-Proschan**
ACT0	0.08695652	0.007968127	0.04347826
ACT1	0.08695652	0.007968127	0.04347826
SEN0	0.2173913	0.01992032	0.1086957
SEN1	0.2173913	0.01992032	0.1086957
SEN2	0.2173913	0.01992032	0.1086957
ECU0	0.2608696	0.02390438	0.1304348
ECU1	0.2173913	0.01992032	0.1086957
ECU2	0.3043478	0.02788845	0.1521739
CH0	0.2173913	0.01992032	0.1086957
CH1	0.173913	0.01593625	0.08695652

Table 6.4. Various Importance measures of basic events.

with different functional to architecture mappings. Some components may be added that are more or less reliable than the components that are replaced, or the functionality may change to support a higher degree of redundancy.

6.2.3 Sensitivity Analysis

The Item Toolkit provides results to indicate which basic events contribute the most to the system dependability, as shown in Tables 6.4 and 6.5. This is known as a sensitivity analysis. Sensitivity analysis typically use Importance metrics to calculate the sensitivity of basic events of a fault tree. The results presented in Table 6.4 uses three types of Importance metrics: the *Barlow-Proschan Importance*, the *Birnbaum Importance*, and the *Fussell-Vesely Importance*. The Barlow-Proschan Importance [19] is the probability that the system fails because a critical cut set containing the event fails, with the event failing last. On the other hand, the Birnbaum Importance [30] represents the sensitivity of the system's unavailability with respect to changes in the event's unavailability. The Fussell-Vesely Importance [7] measure indicates an event's contribution to the system unavailability. Changes in unavailability of events with high importance values will have the most significant effect on system unavailability.

In the synthesis-based design flow described in Chapter 2, the results of the analysis enables quick design exploration of the redundancy and reliability tradeoff, as discussed in [23]. The designer can interpret the analysis results and respond by either changing the mapping, replacing architecture components, or re-designing the control algorithm. Fault tree synthesis enables this capability since fault trees can be generated in minutes using the fault tree synthesis tool on a FTDF model.

Basic Event	**Map1**	**Map2**	**Map3**	**Map4**
ACT0	0.043478	0.050154	0.05	9.83E-06
ACT1	0.043478	0.050055	0.025	9.83E-06
SEN0	0.108696	0.124988	0.125	0.247771
SEN1	0.108696	0.10001	0.125	0.00198
SEN2	0.108696	0.124888	0.125	0.247771
ECU0	0.130435	0.149866	0.125	0.248758
ECU1	0.108696	0.10001	0.125	0.00198
ECU2	0.152174	0.149866	0.15	0.248758
CH0	0.108696	0.075231	0.1	0.001481
CH1	0.086957	0.074933	0.05	0.001481

Table 6.5. Barlow-Proschan Importance for basic events on different mappings.

Chapter 7

Summary

7.1 Conclusions

This thesis presented two aspects to a synthesis-based design flow for real-time embedded feedback control systems: a method for facilitating code generation and dependability analysis using fault trees. The overall design flow, along with the fault tree analysis and code generation, are centered around a formal model of computation, FTDF. FTDF allows the integration of various tools and techniques to unambiguously describe a real-time feedback control system, as shown in this thesis. As such, the same FTDF model can be used to analyze the dependability metrics of the system and generate code from the same model. The MoC is key to the synthesis-based design flow since the flow contains a number of steps from specification to deployment, and at each step the designer may wish to capture specific characteristics of the same system under development from different views by analyzing results at each step. The results are then used to provide the designer with hints on how the system may be improved, hence exploring the design space. In particular, the fault tree analysis allows the designer to assess the structural dependencies more formally using a fault tree model, and the code generation can provide feedback of the system behavior on a real target system. Together, along with other verification tools such as a timing analysis tool, the tools enable the designer to explore the cost versus redundancy (for fault coverage) trade-off for a system described in FTDF. FTDF MoC enables these tools since it gives a precise interpretation of how components behave, and as a result, it provides the foundation for fault tree synthesis and software synthesis of FTDF models.

The examples introduced illustrate the benefits of the approaches taken in the fault tree synthesis and software synthesis of FTDF models. The software synthesis process includes an intuitive graphical interface for specifying and generating code from a FTDF model. The interface constructed allows a visual representation of the same model that may be used with other analysis tools with a goal to produce a run-time implementation to verify the behavior of the system in a real environment. Results from a simple example show that the semi-automatic code generation method made it quicker for designer to prototype a FTDF application as compared to a completely manual implementation for each application the designer chooses to implement. The software synthesis method also frees the designer from coding details or from a difficult interaction with coding experts that results in reduced coding errors and shorter design time. Similarly, the fault tree synthesis allowed the analysis of different system mappings through multiple iterations quicker than if the fault tree generation and analysis were done manually. Together the software synthesis and fault tree analysis tools allow the designer to quickly specify, analyze, and deploy a real-time feedback controller, as depicted in Figure 1.1. With this design flow and the tools described, it is believed that the synthesis and analysis of a design through multiple iterations can be performed faster than if the design synthesis, fault tree generation, analysis, and deployment were performed manually. This flow is an interesting approach especially within the automotive domain to enable quick design exploration within the fault tolerant dimension early in the design cycle where design errors can be detected early and the cost of implementation can be reduced if changes in the system or its specifications occur late in the design cycle.

7.2 Future Work

The work presented in this paper can be extended to incorporate various tools under a common framework. Currently, the analysis and code generation tools function as separate entities that are interfaced through predefined text files. One of the primary goals is to integrate all the analysis tools presented in the overall design flow under a single, possibly graphical interface. The plan then is to completely automate the flow under that single interface, hence, the interface will serve as the core of the synthesis-based methodology. Experience with the GME suggests that it could make a good framework for which these tools can be integrated in a visual form. It is flexible so it may incorporate various tools and support model translation. Then, interpreters can be invoked to execute the analysis tools. This allow the use of visual objects in the construction and manipulation of FTDF models. Furthermore, it reduces the errors found when a team of designers work on a model independently. In this case, a team of designers can work on a single design and ensure its consistency between various tools. The Metropolis [2] design environment is currently the target environment for which the tools will eventually be integrated since the models may be refined and the architecture models may be specified more accurately. It does not currently support a graphical interface, however, its command line interface could be used, as well. One key assumption that is made throughout this paper is assuming a fail silent fault model. It was chosen for simplicity and as a demonstration of the design flow capabilities, however, from the experiences described in this paper it is more interesting to work on this set of tools using a non-fail silent fault model. Finally, FTDF addresses the control algorithms that are used to control a physical plant. A direction that is currently being explored is a formal modeling of the plant and integration of such a model with FTDF.

Chapter 8

Appendix

8.1 Communication Network Interface: User Datagram Protocol Implementation

This section presents the code for implementing network communications using the User Datagram Protocol (UDP) over the Internet Protocol (IP) in the current implementation of the runtime platform. The reader is referred to Section 4.3.3 for more information. As mentioned in that section, the communication network interface is modular, and it is constructed to be protocol independent. As a result, the implementation of those services used by the communication network interface should be implemented in a separate C file (and possible header file(s)). This allows the user a choice of network protocols or means of communicating between processes. For example, if the user chooses to use serial port to communicate between computers, then the protocol specific code used to implement the standard communications services that are used by the network interface in the runtime platform should be implemented in a separate file, like the one shown below. It might be a choice of the designer to use Unix pipes and First-In-First-Out (FIFO) queues to implement communications. If so, then the function calls specific to pipes and FIFOs should be implemented in a file like the one below.

First, the header file for this particular implementation is given.

```
/* Version: 5.1
*  File: ipnet.h
*  Last Updated:  11/12/03
*
*  This is the network API header file. It consists of
*  a library of functions to create a communication
*  channel between multiple computers using UDP/IP
*  socket networking.
**************************************************************/


//General Headers
#include <stdio.h>
#include <string.h>

//Network Headers
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>


#ifndef IPNET_H
#define IPNET_H


//Required common data struct and load function
//These are the standard services that should be
//provided to the runtime platform.

#ifndef SERVICES_
#define SERVICES_
typedef struct services_ {
void (*load) (struct services_ *);
int (*servInit) (char *, int);
int (*sendInit) (char *, int);
int (*serv) (int, char *, int); //an integer handle, ptr to recv buf, size of buf
int (*send) (int, char *, int); //an integer handle, ptr to send buf, size of buf
void (*stop) (int); //closes a socket
} services;
#endif

//Entry function (Required: "[prefix]Load")
extern void ipnetLoad (services *functs);

#endif
```

The source file that compiles with the above header file is given below.

```
/* Version: 5.1
*  File: ipnet.c
*  Last Updated:  11/12/03
```

```
*
*  This is the network API source file. It consists of
*  the definition of the API found in ipnet.h.
*  Note: Currently, these functions only
*  supports UDP over IPv4, in particular.
*  Ideally, the user or designer of the runtime platform
*  may implement other protocols that provide the standard services
*  used by the Communication Network Interface.
*************************************************************/


//
//Headers
//

#include "ipnet.h"

//
//Globals
//

int serversCreated = 0;
int clientsCreated = 0;

//
//Source
//

int udpSock () {
//create UDP socket
//returns -1 on error or file descriptor
return socket (AF_INET, SOCK_DGRAM, 0);
}

int bindSock (int sockfd, char *ipAddr, int ipPort) {

//Binds to ip address/port for receiving data.
//Returns socket bound to address/port for receiving data,
//or -1 on error.

int err;
struct sockaddr_in my_addr;  //my address information
struct hostent *me;

//resolve host
    if ((me = gethostbyname(ipAddr)) == NULL) {
        printf ("ERROR: IPNET: Get host by name failed.\n");
        return -1;
    }
memset (&my_addr, 0, sizeof(my_addr)); //zero out the struct
        my_addr.sin_family = AF_INET; // host byte order
        my_addr.sin_port = htons(ipPort); // short, network byte order
my_addr.sin_addr = *((struct in_addr *)me->h_addr);
printf ("DEBUG: IPNET: Server on %s:%d\n", ipAddr,ipPort);

err = bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
if (err == -1) return -1;
else return sockfd;
}

int setAddrSock(char *ipAddr, int ipPort) {

//??
//Inputs:
// ipAddr, 32-bit address
// ipPort, port
//Returns:
// a connected socket or -1 on error

    int sockfd, err;
    struct sockaddr_in my_addr;    // my address information

//create UDP socket
if ((sockfd = udpSock()) == -1) {
printf ("ERROR: IPNET: Creating socket.\n");
return -1;
}
//bind socket to address
if (bindSock (sockfd, ipAddr, ipPort) == -1) {
        printf("ERROR: IPNET: Binding soccket.\n");
        return -1;
    }
return sockfd;
}


//Required functions for interfacing to ukernel

int ipnetServInit (char *ipAddr, int ipPort) {

//Initialization of Server function.
//Input:
//Return:
//  a handle that is used in the other functions to
//  receive data, send data, close the connection.
//NOTE: this function should return a handle used by
```

```
//the recv functions for this channel interface.
//In the case that the handle is not used in other functions, return -1.
//The handle is a socket or positive number to identify a server.

int servSock;

servSock = setAddrSock (ipAddr, ipPort);
serversCreated++;
return servSock;
}

int ipnetSendInit (char *ipAddr, int ipPort) {

//Initialize client function.
//Inputs:
//  32-bit ip address and port
//Returns:
//  socket connected to client
//NOTE: this function should return a handle used by
//the send functions for this channel interface.
//In the case that the handle is not used in other functions, return -1.
//The handle is a socket or positive number to identify a client.

int clientSock;

clientSock = setAddrSock (ipAddr, ipPort);
clientsCreated++;
return clientSock;
}

int ipnetServ (int sockfd, char rBuf[], int rBufSz) {

//Ip receive function.

int rsize; //size of received data
int addr_len;
struct sockaddr_in their_addr;  //sender's address info

addr_len = sizeof (struct sockaddr);
rsize = recvfrom (sockfd, rBuf, rBufSz, 0,
(struct sockaddr *)&their_addr, &addr_len);
return rsize;
}

int ipnetSend (int sockfd, char sBuf[], int sBufSz) {

//Ip send function.
int ssize;
```

```
struct sockaddr_in their_addr;  //sender's address info
ssize = sendto(sockfd, sBuf, sBufSz, 0,
(struct sockaddr *)&their_addr, sizeof(struct sockaddr));
return ssize;
}

void ipnetStop (int sockfd) {

//Closes a socket.
close (sockfd);
printf ("\nIPNET: Terminated socket %d.\n", sockfd);
}

void ipnetLoad (services *funct) {

//Initializes the services struct for an interface.
//Here, user sets the pointers to the standard
//functions used by the ukernel.

funct->load = ipnetLoad;
funct->servInit = ipnetServInit;
funct->sendInit = ipnetSendInit;
funct->serv = ipnetServ;
funct->send = ipnetSend;
funct->stop = ipnetStop;
}
```

Bibliography

[1] A. Bakay G. Karsai J. Garrett C. Thomason IV G. Nordstrom J. Sprinkle A. Ledeczi, M. Maroti and P. Volgyesi. The generic modeling environment. In *Proc. of the Workshop on Intelligent Signal Processing*, Budapest, Hungary, May 2001.

[2] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe. Modeling and designing heterogeneous systems. *Technical Report 2001/01 Cadence Berkeley Laboratories*, November 2001.

[3] M. Barborak, M. Malek, and A. Dahbura. The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, 25(2):171–220, June 1993.

[4] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The Synchronous Language Twelve Years Later. *Proc. of the IEEE*, 91(1):64–83, Jan. 2003.

[5] C. Lavarenne C. Dima, A. Girault and Y. Sorel. Off-line real-time fault-tolerant scheduling. In *Euromicro Workshop on Parallel and Distributed Processing*, Mantova, Italy, February 2001.

[6] B. A. Davey and H. A. Priestley, editors. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[7] Item Software Inc. Toolkit suite. http://www.itemuk.com/, 2003.

[8] W. Kernighan and D. M. Ritchie. In *The C Programming Language*. Prentice Hall, 1998.

[9] J. Kim and I. Lee. Modular code generation from hybrid automata based on data dependency. In *Proc. of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, Washington, D.C., May 2003.

[10] Kevin J. Sullivan Kiran Kumar Venmuri, Joanne Bechta Dugan. A design language for automatic synthesis of fault trees.

[11] H. Kopetz and D. Millinger. The transparent implementation of fault tolerance in the time-triggered architecture. In *Dependable Computing for Critical Applications*, San Jose, CA, 1999.

[12] Howard E. Lambert. Use of fault tree analysis for automotive reliability and safety analysis. *Computer*, 33(9):18–26, 2000.

[13] L. Lamport. In *LaTeX: A document preparation system*, Reading, Massachusetts, 1986. Addison-Wesley.

[14] E. A. Lee and D. G. Messerschmitt. Synchrounous data flow. In *Proc. of the IEEE*, volume 79, September 1987.

[15] E. A. Lee and S. Neuendorffer. Classes and subclasses in actor oriented designs. In *Proc. of the Conf. on Formal Methods and Models for Codesign (MEMOCODE)*, June 2004.

[16] E. A. Lee and T. M. Parks. Dataflow process networks. In *Proc. of the IEEE*, volume 83, pages 773–801, May 1995.

[17] The MathWorks. Matlab. http://www.mathworks.com/, 2005.

[18] V. V. S. Sarma N. Viswanadham and M. G. Singh. In *Reliability of Computer and Control Systems*, volume 8. North Holland, 1987.

[19] B. Natvig. Reliability analysis: Encyclopedia of acturial science. Technical Report 4, University of Oslo, Department of Mathematics, September 2002.

[20] OMG. Uml documentation. http://www.omg.org/technology/uml, 2005.

[21] C. Pinello, L. Carloni, and A. Sangiovanni-Vincentelli. Fault-tolerant deployment of embedded software for cost-sensitive real-time feedback-control applications. In *Proceedings of Design Automation and Test in Europe*, Paris, February 2004.

[22] M. L. McKelvin J. Sprinkle C. Pinello and A. Sangiovanni-Vincentelli. Fault tolerant data flow modeling using the generic modeling environment. In *Proc. of the IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 229–235, Greenbelt, Maryland, April 2005.

[23] Joe Wysocki Rami Debouk. Redundancy and reliability tradeoffs for safety/mission critical systems. *International Systems Safety Conference*, 2004.

[24] A. Rubini and J. Corbet, editors. *Linux Device Drivers*. O'Reilly, second edition, 2001.

[25] A. Sangiovanni-Vincentelli. Circuit simulation. T. J. Watson Research Center, New York. IBM.

[26] Elmar Dilger Stefan Benz and Klaus D. Mller-Glaser Werner Dieterle. A design methodology for safety-relevant automotive electronic systems. *SAE*, (9):1–14, 2004.

[27] R. Stevens. In *Unix Network Programming: Interprocess Communications*, volume 2. Prentice Hall, 1999.

[28] Esterel Technologies. Esterel studio. Available at: http://www.esterel-technologies.com/products/esterel-studio/overview.html, 2005.

[29] N. H. Roberts W. E. Vesely, F. F. Goldberg and D. F. Haasl. Fault tree handbook. Number NUREG-0492. Division of Technical Information and Document Control, January 1981.

[30] J. Loman W. Wang and P. Vassiliou. Reliability importance of components in a complex system. In *Reliability and Maintainability Symposium*, Los Angeles, California, USA, January 2004.

[31] David Parker Yiannis Papadopoulos. A method and tool support for model-based semi-automated failure modes and effects analysis of engineering designs. *Proc. of the IEEE*, 79(9):1305–1320, September 1991.