A Formal Approach to Fault Tree Synthesis for the Analysis of Distributed Fault Tolerant Systems

Mark L. McKelvin, Jr.*, Gabriel Eirea*, Claudio Pinello†, Sri Kanajan†, and Alberto L. Sangiovanni-Vincentelli*

*Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720

†General Motors Berkeley Lab
Berkeley, CA 94703
mckelvin@eecs.berkeley.edu, geirea@eecs.berkeley.edu,
claudiopinello@cal.berkeley.edu, sri.kanajan@gm.com, alberto@eecs.berkeley.edu

ABSTRACT

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 (for instance in automotive applications). In this paper, we present a novel technique for constructing a fault tree that models how component faults may lead to system failure. The fault tree enables us to use existing commercial analysis tools to assess a number of dependability metrics of the system. Our approach is centered on a model of computation, Fault Tolerant Data Flow (FTDF), that enables the integration of formal verification techniques. This new analysis capability is added to an existing design framework, also based on FTDF, that enables a synthesis-based, correct-by-construction, design methodology for the deployment of real-time feedback control systems in safety critical applications.

Categories and Subject Descriptors

J.6 [**Computer-Aided Engineering**]: Computer-aided design (CAD); C.4 [**Performance of Systems**]: Fault tolerance, Reliability, availability, and serviceability

General Terms

Algorithms, Design, Reliability

EMSOFT'05, September 19–22, 2005, Jersey City, New Jersey, USA.

Keywords

Fault tree construction, system analysis, design exploration, feedback control applications

1. INTRODUCTION

The aerospace and automotive industries are increasingly using electronic hardware and software content that must be designed to be safe and available despite faults that may occur. Hence, the recent strong interest in these industrial sectors to use fault tolerance as a way of coping with strict reliability requirements. Fault tolerance is an attribute that ensures the system remains operational in the presence of a particular set of faults. The formulation of the problem often relates to fault tolerance when the entire functionality of the system has to be guaranteed when faults occur. We focus on cases where only a subset of the functionalities has to be guaranteed while others are not crucial to the safety of the operation of the system. In this case, we must make sure that this subset is operational under the potential faults of the architecture. This case is of interest in the automotive domain.

Typically, fault tolerance is achieved by duplicating somewhat blindly hardware and software components, a solution that is often more expensive than needed especially when we consider the limited functionality requirement. In [13], the problem of introducing redundancy automatically to meet dependability and timing requirements was addressed. An interactive design methodology was proposed to explore the redundancy/cost trade-off. The flow was founded on a new Model of Computation (MoC), *Fault Tolerant Data Flow (FTDF)*, that enables the use of formal techniques for synthesis and validation of the implementation. The central part of that flow is a set of automatic synthesis techniques that process simultaneously the algorithm specification, the characteristics of the chosen execution platform, and the corresponding *fault model*. This information is used to (1) automatically deduce the necessary software process replication, (2) distribute each process on the execution platform, and (3) derive an optimal scheduling of the processes on each electronic control unit (ECU) to satisfy the overall timing

and fault tolerance constraints. Together, replication, mapping, and scheduling result in the automatic deployment of the embedded software on the distributed execution platform. This paper is motivated by the following scenario. If the designer explores two alternative solutions of comparable cost and both *meet* the fault tolerance and timing constraints, can we assess which of the two has a better fault tolerance coverage? In other words, we want to refine our ability to analyze the synthesized deployment in the dependability dimension. In this paper, we propose the use of fault tree analysis [16] to achieve this goal.

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 ways the event may occur in the system. The top-level event of a fault tree is an abstraction of a system failure or malfunction, and it represents the root of the fault tree. FTA uses fault trees as the data structures to represent the relationship between potential hazards and their influence on system reliability. *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 [15]. 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, typically they are constructed manually by subject matter experts. For complex hardware/software systems, this process can be tedious and time consuming. The construction of fault trees, in the general case, requires knowledge of the system behavior and components used to implement the system. However, since our approach is based on FTDF, the task of constructing fault trees can be formalized. In fact, FTDF also 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, corresponding to the synthesized deployment.

There has been prior work in the area of fault tree synthesis and design flows for safety critical systems such as [12] [14] [7]. The work most closely related to this paper is [12], 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.

Our proposal differs in the sense that we have a formal model of computation of the functionality 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. Our 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. The general underlying methodology also differs in the sense that we account for the timing and performance constraints of the architecture on the functionality during the mapping phase.

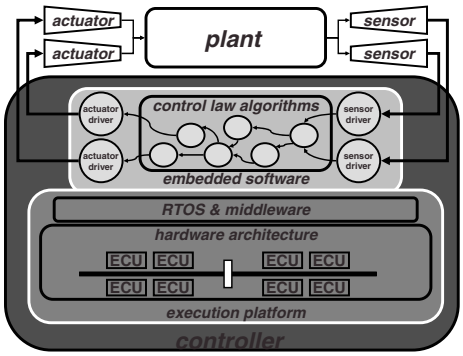

Figure 1: A real-time control system.

In [14], focus was given on how to create a descriptive modeling language (RIDL, Reliability embedded Design Language) that can enable fault tree synthesis. This gives a natural modeling environment for reliability engineers to work and also removes the ambiguity in interpretation of component reliability requirements. Primitives such as single point failure components, support component, repeated components are available in the environment to encode a variety of systems from a reliability stand point. Since these primitives in the modeling language have precise semantics, a fault tree can be synthesized. This work differs from our methodology in two aspects: (1) The language already assumes a mapped system as the input design while in our methodology there is a formal separation of functionality and architecture. This is an issue especially in an automotive domain since reuse of functional models across different architectures is a major requirement. (2) The RIDL language assumes a specific model of computation that is imposed on the mapped system rather than on just the functionality.

The rest of the paper is organized as follows. In Section 2 we briefly present the formulation of the synthesized deployment. For a detailed description of the synthesis algorithm, the reader is referred to [13]. In Section 3 we present FTDF and some of the basic terminology used to describe FTDF graphs. In Section 4 we formulate the fault tree synthesis problem and illustrate how the specification, along with the synthesized deployment, can be used to synthesize a fault tree for the dependability analysis. In Section 5 we apply these techniques on a simple inverted pendulum controller example. Finally, in Section 6 we offer some conclusions.

[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 this version of the implementation, we assumed the architecture can only fail silently.

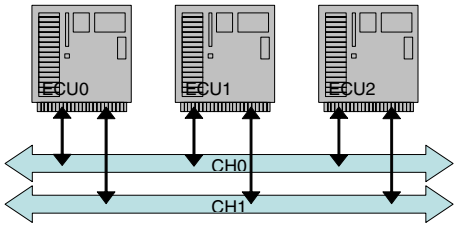

Figure 2: A simple platform graph.

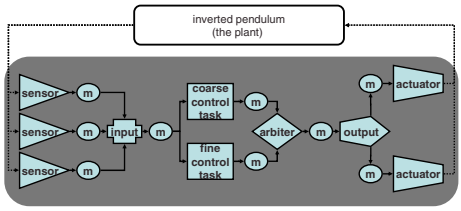

Figure 3: Controlling an inverted pendulum.

2. THE PROPOSED DESIGN METHODOLOGY

For the sake of simplicity, in the fault model that is used in this paper we assume *fail silence*: components either provide correct results or do not provide any result at all. Recent work shows that fail silent platforms can be realized with limited area overhead and virtually no performance penalty [1]. The fail silence assumption can be relaxed if invalid results are detected otherwise, as in the case of CRC-protected communication and voted computation [4]. In our work, fail silence allows us to validate our fault tree synthesis method with high confidence. Furthermore, we use *software replication* to achieve fault tolerance: critical routines are replicated statically (at compile time) and executed on separate ECUs and the processed data are routed on multiple communication paths to withstand channel failures.

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 guarantees that for each fault scenario, or *failure pattern* [5], the execution of a corresponding subset of the actors in FTDF graph, $\mathcal{G}$ must be guaranteed. Figure 3 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 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 $T_{\max}$ (our real-time constraint). Moreover, the control algorithms must be executed in spite of the possible platform faults.

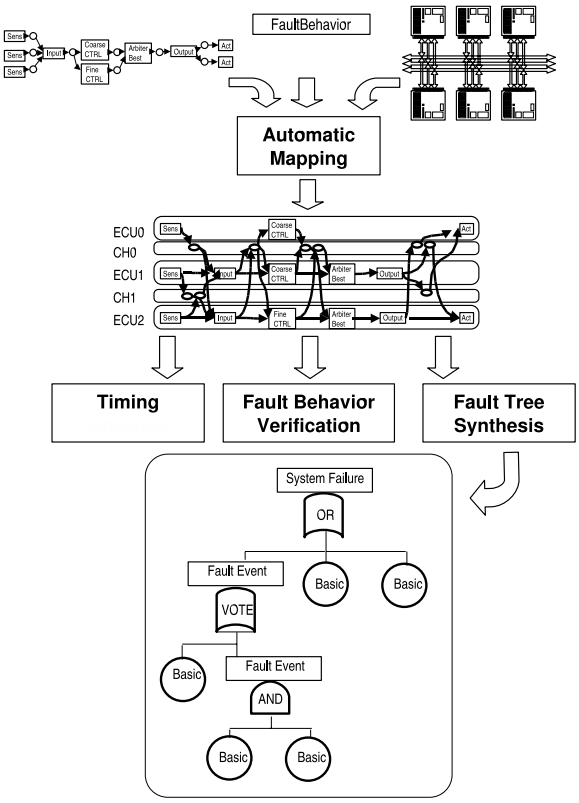

Figure 4: Proposed Design Flow.

Figure 4 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 deployment. Finally, a verification tool checks whether the fault tolerant behavior and the timing constraints are met.

If more than one solution is found, we would like to assess precisely their dependability to correctly explore the fault tolerance/cost trade-off. In this paper, we propose a synthesis method to automatically generate a fault tree model for each of the correct deployments. Existing fault-tree analysis tools, such as the Item Toolkit [6], can then be used to analyze the deployments and score them on the dependability dimension.

3. FAULT TOLERANT DATA FLOW

Fault Tolerant Data Flow (FTDF) is a synchronous [3] model of computation (MoC) where every actor executes once per iteration, satisfying the precedence order dictated by the data dependencies. Then, the next iteration may start. This section reviews the fundamental components of an FTDF model: tokens, actors, and communication media. An FTDF graph provides the structural dependencies amongst components in a FTDF model. As described in Section 3.3, FTDF applications are *fault model independent*.

3.1 Tokens

Tokens are encapsulations of data that is transferred between functional components in a FTDF graph. Since this paper focuses on the construction of fault trees and tokens are specified as part of the operational semantics of FTDF, the reader is referred to [13] for a detailed assessment.

3.2 Actors

In an actor-oriented design framework [9], 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. Here this abstraction is referred to as a *medium*. Actors also contain a *firing rule* and a *firing function* that specifies 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.

In FTDF, actors are typed. An 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, or tokens, are present. We represent this 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[4]. This is the typical firing rule found in other data flow languages [8] [10]. 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:

$$U = \{(*, *, *), (\perp, *, *), (*, \perp, *), (*, *, \perp)\}, \qquad (1)$$

where $\perp$ denotes the absence of a value. The execution semantics of FTDF dictates that, in 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* actors may be present in a FTDF graph. This component stores results produced during the current period of an 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. More precisely, the medium provides the correct merged result or an invalid token if no correct result is determined.

Assuming *fail-silence*, merging is a selection of *any* valid results; assuming *value errors* majority voting is necessary to perform the merge; assuming Byzantine faults we need multiple rounds of voting in order to merge the data (see the consensus problem [2]). 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). During the fault tree analysis, we abstract these implementation details about communication media distribution, and use a simplified characterization of communication media. Using communication media, actors always receive exactly one token per input (possibly *invalid*) and the application behavior is independent of the type of platform faults.

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

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

$$A = A_S \cup A_{Act} \cup A_I \cup A_O \cup A_T \cup A_A \cup A_M, \qquad (2)$$

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
- *Sensor* actors have no inputs from other actors, and *Actuator* actors have no outputs to other actors.

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. This basic condition is a motivation for *replica determinism*. Replica determinism in [13] states that if two actors in $\mathcal{R}(v)$ fire, they produce identical results.

4. FAULT TREE SYNTHESIS

In this section, an algorithm for automatic fault tree construction from a redundant mapped FTDF graph is formulated and described. The static fault tree generated is represented in a series of Boolean relationships. The fault tree describes 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.

4.1 Problem Formulation

A redundant mapped FTDF graph $\mathcal{L}_{FT}$, is generated by the synthesis algorithm described in [13], and it is given as

$$\mathcal{L}_{FT} = (L_{V_{FT}}, L_{E_{FT}}), \qquad (3)$$

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

$$f_{v_g} : B^N \rightarrow B \tag{4}$$

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 [15].

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

$$\mathcal{F} = (F_G, I_B, F_N), \tag{5}$$

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.

4.2 Fault Tree Construction 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 hardware 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. $l_{a_i} = (p_i, a_i) \in L_{V_{FT}}$,
    $e_i =$ **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 an abstraction of a fault in the electro-mechanical hardware of the sensor or actuator[5]. 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.

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

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 between actors on the same ECU via shared memory or if it is 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 an event for each edge l_k, and it adds each event to an *OR* gate. An event created at this point models the fact that remote data from actor instance l_k is not delivered to actor instance $l = (p, a)$ when 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*. The *InputFaultEvent* event (e_j) is then composed of the logical *AND* of the fault events of the various replicas generating the input. The choice of the *AND* composition derives from the fail silent assumption; changing the fault model will impact how these events must be composed. 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. 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.

4.3 Algorithm Complexity

A mapped FTDF graph is given as input to *DevelopSubTree*. Actors in the mapped graph are traversed in a depth first traversal scheme from actuator actors to sensor actors. As each actor in the mapped graph is visited once through recursion of *DevelopSubTree*, a subtree of events is created and stored in memory. Each subtree contains levels of fault events that contribute to the failure of that actor instance. The first level corresponds to the failure of the actor instance in the mapped FTDF graph. Each actor instance will have a first level fault event, therefore, its memory complexity is $O(1)$. The second level is a combination of three fault events that must occur for the actor instance to fail. Fault events at the second level are an actor not receiving enough input tokens to fire, the ECU for which the actor instance is mapped fails, or the actuator hardware fails to update the actor. The second level results in a memory complexity of $O(1)$. The third level contains events for determining when an input to the actor instance is faulty, as in the case where an actor instance cannot fire due to the lack of input tokens. Fault events on the third level depend on the number of inputs to an actor instance, and they are created and stored in the first recursive call to *DevelopSubTree* on each input with a memory complexity of $O(D)$, where D is the average number of inputs per actor instance in the mapped FTDF graph. The fourth level constructs subtrees that determine how fault events occur for the source actor that feeds into the actor instance. The second call to *DevelopSubTree* occurs at this point, and since a subtree of fault events are developed for the inputs to the source actor, it has a complexity of $O(D)$. At the point where the algorithm

[5]Since we assume fail silence, we are only concerned with the actor producing or not producing a token. This is a high level abstraction, and we do not consider the internal mechanics of a sensor or actuator device.

reaches a sensor actor, the recursion ends since sensor actors have no input events to further develop. Furthermore, in the case that an actor is replicated (i.e. it contains the same actor dependencies, but located on a different ECU), the subtree of the replicas will be the same, and each subtree is stored individually. The resulting memory complexity for developing a subtree for each actor instance in the mapped graph is $O(D^2)$. Given a total number of M actor instances in the mapped graph, the resulting memory complexity is $O(MD^2)$. The execution time complexity for the algorithm is $O(D^{2M})$ since each call to *DevelopSubTree* is $O(D^M)$. The performance of the algorithm is computationally expensive and highly dependent on the number of actors and communication channels in the mapped FTDF description.

5. CASE STUDY

In this section we present the case of an inverted pendulum control system, a common example that is used in the feedback control system domain. Since manually constructed fault trees are highly dependent on subjective factors such as specified failure modes, system topology, and the analyst's understanding of the system, a complex example for this work would be difficult to evaluate. Our example is simple enough to validate the quality of the fault tree produced by our synthesis algorithm by inspection. The controller is described by an FTDF graph that has a topology similar to that of automotive subsystems that contain feedback controllers, as shown in Figure 3. It consists of three position Sensors, one Input actor that performs sensor integration and assesses the current pendulum position, two different Task actors that represent two controllers (coarse and fine) that require different computing power, one Arbiter that selects the output of one of the controllers (the fine one, whenever available), and an Output actor that directs the control action to two different Actuators. The execution platform is given in Figure 2 and consists of three ECUs and two communication channels. 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 redundant deployment consist of:

- execute the entire algorithm in absence of faults (default behavior)
- in the presence of a single ECU fault, guarantee the execution of the critical 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",
- the Output actor "OUT".

In particular, in order to fire, actor "IN" requires that at least two of the Sensors deliver their data to it. The Arbiter actor can fire with one of its two inputs present.

As a result, the synthesis tool performs the redundant mapping described in Table 1. It is worth noting that the fine controller "FUNf" is not replicated, because it was not specified as part of the critical set. This mapping is guaranteed by construction to tolerate single ECU failures. However, there is no finer 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 how and why the system may fail. The fault tree is then analyzed by the Item Toolkit [6], a commercial tool distributed by Item Software Incorporated. Among many analyses that can be performed, we present here the cutset analysis, the dependability analysis, and the sensitivity analysis.

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 1: Redundant mapping of the pendulum example.

5.1 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 2. The name of the basic faults correspond to hardware faults in Sensors, Actuators, ECUs, or communication channels. Based on Table 2, it is clear that no single ECU failure leads to the system failure (assuming the top-level event is the failure of all actuators in the given system mapping). Moreover, no single channel failure leads to a system failure. Note that channel failures were not part of the fault behavior specified to drive the deployment synthesis algorithm.

5.2 Dependability Analysis

The dependability 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, we assume a system lifetime of 5000 *hours*, all basic events have the same reliability, and we assume a classic exponential distribution, where the MTTF of a basic event is 2000 *hours* and the MTTR of a basic event is 8 *hours*. The Item Toolkit returns a MTTF of 11015.81 *hours* for the system.

Based on the simple specification, we could expect a reliability higher than that of a single component failing ($MTTF$ *(system)* $>$ 2000 hours). We could also recognize that the solution provides a dual redundant system in its weakest points. Hence, assuming no repair, we could expect $MTTF(system) \geq 1.5 \cdot MTTF(baseevent) = 3000$ *hours*. Having a fault tree now allows experimenting with different mixes of more or less reliable components to explore the design space [18], as shown below.

5.3 Sensitivity Analysis

The Item Toolkit also provides various Importance metrics for systems under analysis. The Importance metrics

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 2: Cutsets for the pendulum example.

considered in our analysis include the Barlow-Proschan Importance [11], Birnbaum Importance [17], and the Fussell-Vesely Importance [6]. The Barlow-Proschan Importance metric is the probability that the system fails because a critical cut set containing the event fails, with the event failing last. The Birnbaum Importance represents the sensitivity of the system's unavailability with respect to changes in the event's unavailability, and the Fussell-Vesely Importance indicates an event's contribution to the system unavailability. As shown in Table 3, our results, indicate which basic events contribute the most to the system dependability based on the given Importance metric. From the results listed in the table, one can clearly see that ECU2 contributes most to the system's dependability, and the result is consistent with our expectations since ECU2 contains more actors than ECU0 or ECU1, thus making ECU2 a highly critical component in the given system mapping shown in Table 1. Table 4 provides the importance values for the basic events in each of four different system mappings. In the table, the first column is the basic events. The last three columns are the Barlow-Proschan Importance metrics for each mapping.

5.4 Design Exploration

The results of the analysis show that the highest contributors to system failure are the ECU faults, then the sensors faults, the channels faults, and the least impact is due to actuator faults. Our first attempt at improving system MTTF would obviously be to improve reliability of the ECUs, and in general of each of the components. This would clearly result in longer MTTF for the system.

In this paper we are more interested in exploring how different deployments may affect dependability, without changing the execution platform. For this reason, we synthesized the following three additional deployments:

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 3: Various Importance metrics of base events.

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 4: Barlow-Proschan Importance for basic events on different system mappings.

1. Map 2. We request that all critical actors be executed also in the presence of any two ECU faults, i.e. corresponding to failure patterns $\{ECU0, ECU1\}$, $\{ECU1, ECU2\}$, $\{ECU0, ECU2\}$.

2. Map 3. In addition to requirements in Map 2, we request that all critical actors be executed also in the presence of any single-channel fault, i.e. corresponding to failure patterns $\{Channel0\}$, $\{Channel1\}$.

3. Map 4. We use the same requirements as in Map 3, but we change the sensor fusion algorithm so that now it can correctly estimate the pendulum position also using a single measurement from the sensors. Its firing rule is now $U = \{(*,*,*), (\perp,*,*), (*,\perp,*), (*,*,\perp), (\perp,\perp,*), (*,\perp,\perp), (\perp,*,\perp)\}$.

The deployment synthesis algorithm cannot meet the requirements for Map 2 and Map 4, for example because the failure of two ECUs makes it impossible to read two of the sensors, so the sensor fusion actor cannot fire and none of the following actors can fire. Nonetheless, the mapping tool introduces in the deployment more redundancy in the execution of actors, essentially there are now three replicas of each of the critical actors. Also, in Map 3 and Map 4, communication is more redundant. After generating the fault trees for the three deployments, and analyzing them in the Item Toolkit, we obtain the results as in Table 5.

Table 5 shows that the additional redundancy improves the MTTF only marginally, whereas the use of a more robust sensor fusion algorithm yields much better results in this example. The end-to-end latency is the result of a timing analysis for the four deployments and indicates the la-

Designs	System MTTF (Hours)	Number of Minimal Cutsets	Latency (Micro-seconds)
Map 1	11011.81	23	260
Map 2	12652.23	23	260
Map 3	12663.58	20	280
Map 4	62752.8	15	280

Table 5: For each deployment we plot the system MTTF, the number of cut-sets, and the end-to-end latency.

tency from reading the sensors to updating the actuators. From the results, it is clear that having more replicas to be executed on the same execution architecture leads to longer latencies. This information allows the system designer to trade-off between the desired replication and required latency through the controller.

While more exploration directions are possible, the focus of this paper is to show how to automate the analysis of the synthesized system deployments. To this end, it is worth noting that we synthesized each additional deployment and each fault tree in less than 1 minute. Then, the Item Toolkit can generate the analysis results in less than 2 minutes.

6. CONCLUSIONS

This paper presented an approach to generate automatically a fault tree for a given system. The approach is based on a formal MoC, FTDF, introduced to synthesize fault tolerant implementations. Since the model can give also a precise interpretation of how components fail and the manifestation of that failure in the functionality, it can be used to provide the foundations for the fault tree synthesis approach. In the automotive domain, subsystem suppliers should provide Original Equipment Manufacturer (OEM) parts with guaranteed properties. Having a formal representation of the fault tolerance requirements and properties allows the OEMs to verify that their suppliers are compliant. We introduced a simple case study to illustrate the benefits of our approach: an inverted pendulum controller. The inverted pendulum case study was simple enough to validate the synthesis algorithm yet complex enough to exercise our claims on the methodology which was to make the design process for safety critical systems faster and formal.

With our flow, we believe that the synthesis and analysis of a design through multiple iterations can be performed more efficiently than if the design synthesis, fault tree construction, and analysis were done manually. In fact, the flow from the replication synthesis to the generation of a fault tree for input into the Item toolkit was completely automated. This flow is an interesting approach especially within the automotive domain to enable quick exploration within the fault tolerant dimension early in the design cycle.

Furthermore, this paper focused on the contributions of fault tree synthesis in the design flow from a reliability standpoint. The core of the synthesis engine is a computationally expensive algorithm that traverses a mapped FTDF graph of the system under analysis. Our future work will address this issue by reducing the memory requirements imposed by the fault tree construction algorithm. The current implementation of our fault tree synthesis algorithm assumes fail silence, however, future work will relax this assumption and account for a more dynamic array of potential faults that commonly occur in practical systems. Furthermore, we plan to extend this approach by accounting for dynamic elements in the fault tree to capture sequential faults and run-time fault mitigation strategies. Another interesting aspect that is worth exploring is the generalization of the math formulation for the synthesis engines, both for deployment and for fault tree construction, so that the flow can also cope with different failure modes for the hardware components or different formal MoCs for the functionality.

7. ACKNOWLEDGEMENTS

We would like to acknowledge the constructive discussions with Joe Wysocki and Rick Clemons from HRL Laboratories as well as General Motors Research staff, specifically Rami Debouk and Thomas Fuhrman. In particular, HRL offered their expertise in the use of the Item toolkit, Fault Tree Analysis, and Rick Clemons performed all the analyses on the fault trees that we synthesized. Paolo Giusto and Max Chiodo from the General Motors Berkeley Lab were also instrumental in giving critical feedback on the ideas. The inputs from Professor Kurt Keutzer, Alain Girault and Cătălin Dima are also greatfully acknowledged.

This work was supported in part by the Center for Hybrid and Embedded Software and Systems under the National Science Foundation ITR grant CCR-0225610, the Gigascale Systems Research Center under the Microelectronics Advanced Research Corporation grant 2003-DT-660, and General Motors.

8. REFERENCES

[1] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, M. Peri, and S. Pezzini. Fault-tolerant platforms for automotive safety-critical applications. In *CASES*, pages 170–177, San Jose, California, USA, 2003. ACM Press.

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

[3] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous language twelve years later. *Procs. of the IEEE*, 91(1):64–83, Jan. 2003.

[4] F. V. Brasileiro, P. D. Ezhilchelvan, S. K. Shrivastava, N. A. Speirs, and S. Tao. Implementing fail-silent nodes for distributed systems. *IEEE Transactions on Computers*, 45(11):1226–1238, November 1996.

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

[6] I. Item Software. *FaultTree+ for Windows*, volume 8.0. Isograph Limited, 1998.

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

[8] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Procs. of the IEEE*, 79(9), September 1987.

[9] E. A. Lee and S. Neuendorffer. Classes and subclasses in actor oriented designs. In *Procs. of the Conference on Formal Methods and Models for Codesign*

(MEMOCODE), San Diego, California, USA, June 2004.

[10] E. A. Lee and T. M. Parks. Dataflow process networks. *Procs. of the IEEE*, 83(5):773–801, May 1995.

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

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

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

[14] K. K. Venmuri, J. B. Dugan, and K. J. Sullivan. Automatic synthesis of fault trees for computer-based systems. *IEEE Transactions on Reliability*, 48(4):394–402, December 1999.

[15] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. Fault tree handbook. Technical Report NUREG-0492, U. S. Nuclear Regulatory Commission, Division of Technical Information and Document Control, January 1981.

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

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

[18] J. A. Wysocki and R. Debouk. Redundancy and reliability tradeoffs for safety/mission critical systems. In *International Systems Safety Conference*, Providence, Rhode Island, USA, August 2004.