Model-Based Design of Heterogeneous Systems for Fault Tree Analysis

Mark L. McKelvin, Jr.; University of California; Berkeley, California, USA

Claudio Pinello; General Motors; Berkeley, California, USA

Sri Kanajan; General Motors; Berkeley, California, USA

Joseph Wysocki; Malibu, California, USA

Alberto Sangiovanni-Vincentelli; University of California; Berkeley, California, USA

Keywords: fault tree generation, system analysis, system modeling, feedback control applications

Abstract

We introduce a model-based approach to heterogeneous system design that enables the automatic generation of fault trees for analyzing system reliability properties. This approach extends our previous work that addressed the generation of fault trees from a dataflow model. In this new context, heterogeneous systems are composed of interacting discrete-time components, such as an electronic feedback controller, and continuous-time components, such as a plant. More recent work in computer-aided fault-tree generation methods is based on functional models of the system to produce a system fault tree automatically. Yet, most of these approaches were not applied to heterogeneous systems. Furthermore, these approaches continued to rely on intuition to create fault trees. Since in this approach fault tree generation is disjoint from the system modeling, consistency problems may arise when the structure and behavior of the system model is not accurately reflected. Our approach is different since we use a model of the system specified as a set of mathematical equations to derive the system fault modes and ultimately produce fault trees for heterogeneous systems.

Introduction

Increasing complexity of real-time feedback control systems coupled with time-to-market pressures have prompted system engineers and reliability experts to assess the safety and reliability properties early in the design cycle at the system level. In this context, complex engineering systems are heterogeneous. Heterogeneous systems contain components that interact according to rules that may differ from component to component. For example, in an electromechanical system, there are mechanical components that interact according to Newton's laws, and there are electronic components that interact according to Kirchoff's laws. Consequently, different formalisms have been developed to specify system behavior within their respective domains for modeling, simulation, and analysis. For example, in the case of an analog electrical system, electrical circuits are often used to specify the behavior.

Fault tree analysis is used to measure, estimate, and predict dependability characteristics of a system. Dependability includes reliability and safety properties. Fault tree analysis focuses on one undesired event and provides a graphical structure, called a *fault tree*, which illustrates the sequences of events that lead to the occurrence of that undesired event. This paper considers the generation of fault trees, but we must mention that other dependability data structures exist, such as Markov chains, event trees, reliability block diagrams, and Bayesian networks for capturing the reliability properties of a system. A fault tree is in general constructed manually from design specification documents and intuition by highly skilled and experienced subject-matter experts. Due to increased system complexity, constructing fault trees manually can be time-consuming, and the fault tree construction process may introduce inconsistencies with the system intended behavior.

In this paper, a methodology is described that leverages the advances of computer-assisted design tools and fault tree analysis for investigating properties and exploring the design space of complex engineering systems along the reliability dimension. The method specifies a heterogeneous system in a model that may be used to simulate system behavior. From this model, a fault tree reliability data structure is generated. The resulting fault tree is then used as input into existing reliability tools for generating quantitative and qualitative metrics.

Fault Trees

A reliable and dependable system should provide its intended behavior in a correct and timely manner. The inability of a system to provide this performance is referred to as a *failure*. Activities in any system can lead to a defect, omission, incorrectness, or other *flaw* that enters into, or develops within that system. Then, a fault is a flaw, and a set of faults may lead to a system failure. One way to assess a system failure is using a *fault tree*.

Events: Fault events in a fault tree are either *derived events* or *basic events*. A *top event* is an undesirable derived event that one wishes to evaluate in a fault tree. The top event is specified *a priori*, and it is the root of the fault tree. Derived events can always be "explained" (modeled) as a logical combination of other events. *Basic events* are atomic events that may not be derived any further; hence, they are the leaves of the fault tree. In other words, a basic event is an input failure event for the system fault tree. Determined at the discretion of the analyst, basic events typically correspond to the atomic components in a system

Gates: The fault tree relates input and output fault events with logical symbols. Symbols that encode logical relationships are referred to as *gates*. Gates can take as inputs basic events or output events of other gates (derived events). Common gates with Boolean logic semantics include the *AND* and *OR* gates. Other gates may be used in a fault tree, such as a conditional logic relationship. We consider *AND* and *OR* gates only in this paper.

Fault Tree Construction: There is no precise universal method for constructing a fault tree. However, over the years a general procedure to guide fault tree construction has been developed (ref. 1). One of the first steps in constructing a fault tree is to identify a top event and the boundaries of the system structure and behavior. A set of top events and the level of resolution for the analysis are identified. Then, the system analyst determines the top event of interest and all possible events leading to its occurrence. Next, the first level of events that immediately contribute to a top event is linked to it in the fault tree using a logic gate. Then the second level of contributing fault events is identified and linked to the fault tree using logic gates. Finally, this process is repeated until basic events of the system are identified. In practice, fault trees are constructed manually following this procedure.

Related Work

There have been attempts at generating fault trees automatically. Approaches are typically different in the type of input data structure used to systematically generate the fault tree and the way failure modes are attributed to the system components. Failure modes determine how components in a system fail. Fault tree analysis is difficult because the input data structures used for generating the tree are typically not the system design models. Many design models do not contain enough information, such as failure modes and error propagation, to support the fault tree construction. Instead, such information appears in reports that are used to build data structures from which fault trees are generated.

For example, in (refs. 2, 3, 4), labeled directed graphs are constructed to describe explicitly the cause-and-effect relationships between process variables and process environment. The graphs are constructed manually from design documents and knowledge of the system behavior. These graphs are then transformed into a fault tree. In a method developed by Fussell (ref. 5), each component in an electrical system schematic has a small fault tree that is used to embed failure modes of the component. The system fault tree is composed from the individual fault trees of the components. Thus, the approaches above produced data structures that are not easy to build, reuse, and keep consistent with the system design models; fault modes are described by a small set of discrete values, and the intermediate data structures can become complex, limiting applicability for large systems.

More promising approaches focused on integrating enough detail in the models used for designing and simulating the system, for example into block diagram schematics and behavioral models. These models describe both the structural dependencies and the behavior of components in the system. However, many of the methods are domain-specific, and they do not generally apply to a complex system. An example is the method developed by De Vries (ref. 6) for analog electrical circuits which quantifies fault modes. In contrast, methods such as the ones reported in references 7 and 8 address digital systems where each component in the system model is annotated with a small set of discrete fault modes without quantifying the fault. For example, in these approaches, a parameter of a component is qualitatively labeled as “too high” or “too low” and does not quantify the fault in terms of a real value. Furthermore, fault events for the system fault tree are developed from the fault modes that are annotated on the component models.

Our approach is different since:

- the use of a model-based design paradigm maintains the system specification, while providing a sufficiently accurate model of behavior interactions between various physical domains;
- fault trees may be generated from this specification more accurately and with higher confidence than manual methods. Rather than relying on expertise knowledge, fault events are determined by looking at the type of modeling components in the system model;
- our fault tree generation procedure may be applied to synthesizing fault trees automatically for the purpose of design exploration along the reliability dimension.

Model-Based Fault Tree Generation

A model is generally represented as a set of interacting symbols. The set of symbols and rules for interpreting a composition of those symbols is known as *model of computation* (ref. 9). One of the challenges of modeling any system is choosing the right level of abstraction and the model of computation that expresses relevant aspects of a system. Numerous design tools and languages for modeling various abstractions and behaviors of heterogeneous systems include Simulink with Stateflow (ref. 10), Modelica (ref. 11), Ptolemy (ref. 12), and Metropolis (ref. 13).

The approach presented here takes advantage of modeling capabilities in advanced tools to generate a fault tree based on a structural and behavioral model of a heterogeneous system. We have considered the case where a complex engineering system is described as exhibiting discrete and continuous behaviors across different domains (*e.g.*., electrical, mechanical, and digital,). For example, consider an electronic controller interacting with a physical process (or plant) as in Figure 1.

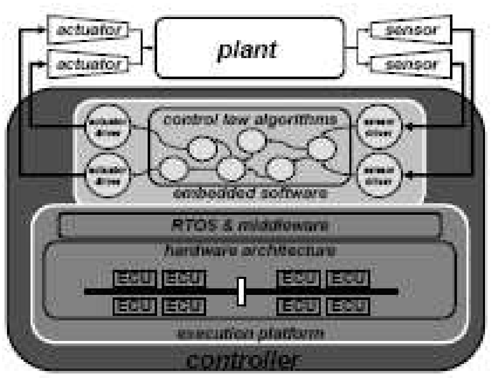


Figure 1- Electronic Feedback Controller and Plant

Electronic Controller: In digital control systems, the outputs of an elementary controller block can be any function of its inputs. Hence, the relationships between inputs and outputs are typically not the transfer and storage of physical energy. For example, the Fast Fourier Transform of a set of physical signals can form an elementary control block. This control block works with (energy-less) information signals, or sequences of data. A dataflow model of computation called *Fault Tolerant Data Flow* (FTDF) was used to model an electronic controller in reference 14. The new method presented in this paper extends the FTDF method to consider the plant during the generation of the fault tree. This allows expressing time-dependent events such as a variable’s rate of change between the plant and the digital controller. This paper focuses on the method for generating a fault tree for the plant. Our method for generating a fault tree for the digital controller can be found in reference 15.

Physical Process: This section details the approach taken in this paper to model the physical process of a complex engineering system for the purpose of systematically generating a fault tree. The elementary objects in physical process are:

- a *dynamical system* that exhibits continuous-time behavior as prescribed by the laws of physics,
- the interconnections between physical objects that carry real physical energy.

The physical process typically encompasses different interacting energy domains, such as mechanical, hydraulic, electrical, and magnetic. The behavior of a dynamical system is characterized by continuous *state variables*, such as velocity, and component *parameters*, such as a vehicle's mass. In particular, we consider the system dynamics, which can be formulated as an initial-value Ordinary Differential Equation (ODE) problem (ref. 12). The general form is given in equations 1, 2, and 3.

$$x' = f(x, u, t) \quad (1)$$
$$y = g(x, u, t) \quad (2)$$
$$x(t_0) = x_0 \quad (3)$$

where, $t \in \mathcal{R}$, $t \geq t_0$, a real number, is continuous time. At any time t, $x(t) \in \mathcal{R}^n$, an n-tuple of real numbers, is the state of the system; $u(t) \in \mathcal{R}^m$ is the m-dimensional input of the system; $y(t) \in \mathcal{R}^l$ is the l-dimensional output of the system; $x'(t) \in \mathcal{R}^n$ is the derivative of x with respect to time t, i.e. $x'(t) = dx(t)/dt$. Given a known input waveform $u(t)$, and initial condition $x(t_0) = x_0$, the solution to system dynamics (eqn. 1) is a state waveform, $x(t)$ in the n-dimensional space such that at all time $t \geq t_0$, the derivative of the waveform, $x(t)$ is given *by* $f(x(t), u(t), t)$.

In general, there are two common specifications for ODEs, the conservation-law model and the signal flow block-diagram model (ref. 16). The conservation-law model generalizes different energy domains into *flow* and *across* variables. Energy conservation laws govern the behavior of those variables in a unified mathematical framework. Examples that use the conservation-law model specification include Bond Graphs (ref. 17) and equivalent analog circuits (ref. 18). ODEs may also be specified through a signal flow graph model. The signal flow graph model is a directed graph where edges represent input or output signals, and nodes compute a function that maps input signals to output signals, as shown in Figure 2.

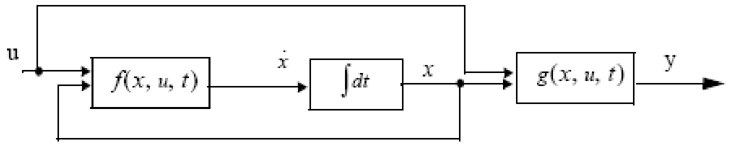


Figure 2 – Conceptual Signal Flow Graph

The signals in Figure 2, u, x', x, and y, are continuous waveforms flowing from one block to the next. Since time is shared by all blocks, it is not considered an input. At any fixed time t, if the values $x(t)$ and $u(t)$ are given, $x'(t)$ and $y(t)$ can be found by evaluating (i.e. simulation) $f(x, u, t)$ and $g(x, u, t)$. For the purpose of fault-tree generation, the signal flow model is of greater use because it provides an intuitive way of capturing the causality relationship between model components, integrating other models of computations, and offering flexibility to the system analyst for modeling different abstraction levels.

System Model Specification: In this paper, we chose to identify four different types of blocks that are commonly used in signal flow graphs and that are useful in our approach to generating fault trees of dynamic systems. Across different design tools, like Ptolemy and Matlab, the syntax of these blocks may differ slightly; however, the mathematical relationship between input and output signals is the same.

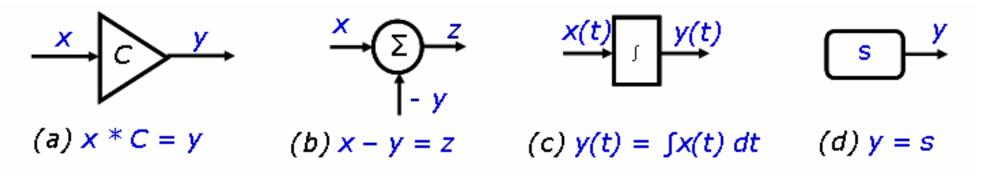


Figure 3 – Signal Flow Graph Symbols

Coefficient blocks: When a system variable is multiplied by a coefficient, this function is represented by a coefficient block, as shown in Figure 3(a). This figure includes the mathematical operation represented by the block and its input and output variables, as do all blocks in Figure 3. The directed edges into and out of the block represent signal flow, and they are not necessarily representative of physical connections. The set of coefficient blocks in the signal flow graph, G, is denoted by, B_C.

Summation blocks: A summation block is used when a variable is equal to the sum of two or more other system variables; the relationship between its output and input signals is shown in Figure 3(b). The set of summation blocks in G is $\mathrm{B_S}$.

Integration blocks: An integration block is used when a system variable is the time-integral of another system variable. Its functional relationship between its input and output signals is shown in Figure 3(c). The set of integration blocks in G is B_I.

Input blocks: Input blocks are system variables that are inputs to the dynamic system. These blocks have no inputs, and they represent external variables to the system. The mathematical relationship between its input and output signals are shown in Figure 3(d). The set of input blocks in G is B_R.

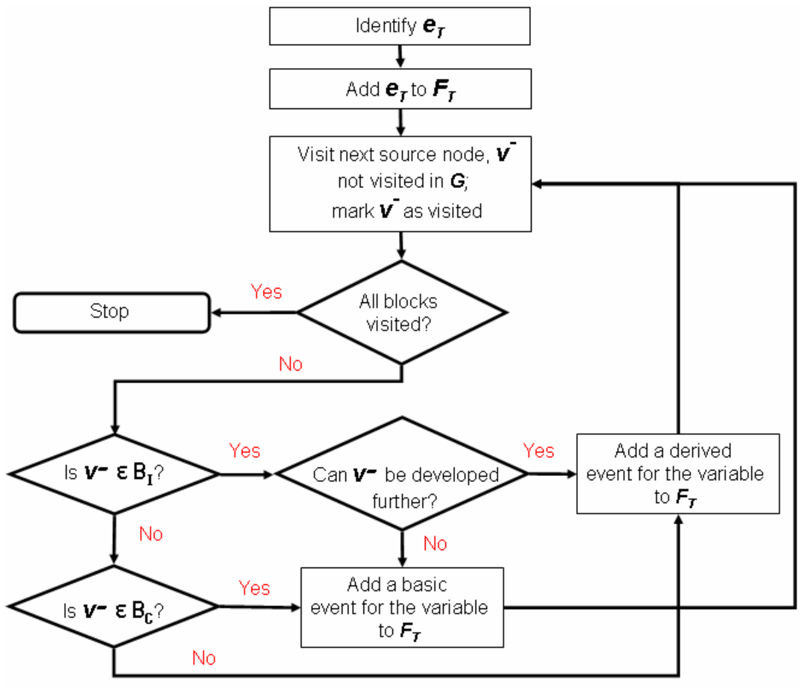
We can formally represent the signal flow graph as a directed graph, $G = (V, E)$, where the set of vertices, $V = B_C \cup B_S \cup B_I \cup B_R$, and a directed edge, $e = (v^-, v^+)$, represent signal flow where $v^- \in V$ is a source node, $v^+ \in V$ is a destination node, and $e \in E$.

Fault Trees and System Models

Fault Model: A fault event, and consequently a fault tree, is a snapshot of the system captured at a specific point in time. In this paper, we only consider the fault model for which any event, $e \in \{0, 1\}$, a Boolean value. Hence, if an event $e = 0$, the event has not occurred at the given point in time, and conversely, if $e = 1$, the event is said to have occurred. Essentially, we enumerate the possible occurrence of faults for each parameter and variable in the model.

No combination of occurrences is considered because with our current fault model we do not quantify faults. Hence, we must assume that a single fault may propagate un-masked to the next level up in our tree. As a result, the corresponding fault tree will only contain *OR* gates. For example, if three forces, (f_1, f_2, f_3), are acting on a point in the system model then a branch in the tree would have an *OR* gate and underneath the *OR* gate there would be three events. The parent event occurs as soon as one of f_1 *OR* f_2 *OR* f_3 fails. Clearly, if the combination f_1 *AND* f_2 is true, so is the formula f_1 *OR* f_2 *OR* f_3, so we do not need to use any *AND* gates.

Fault Tree Generation Procedure: This section describes how to traverse systematically the system model, G, and generate fault events for the system fault tree, F_T. The general steps are outlined in Figure 4



Identify e_T
Add e_T to F_T
Visit next source node, v^- not visited in G; mark v^- as visited
All blocks visited?
Yes
Stop
No
Is $v^- \varepsilon B_I$?
Yes
Can v^- be developed further?
Yes
Add a derived event for the variable to F_T
No
No
Is $v^- \varepsilon B_C$?
Yes
Add a basic event for the variable to F_T
No

Figure 4 – Systematic Fault Tree Generation Method

First, we select a specific edge on the system model to define the top event, e_T. The signal of the selected edge is placed at the root of the fault tree as the top event. G is traversed backward to the next source node v^-. If v^- is an integral block, a derived event that describes a rate-dependent fault is added to the tree, e.g. "dx/dt faults". If the rate-dependent fault cannot be developed further, it becomes a basic event. If v^- is a coefficient block, then each parameter describing the coefficient block is added as a basic event to the tree. For example, if a coefficient block is described as a function of two constant parameters m and c, then two events would be entered into the tree under an *OR* gate as "*m* faults" and "*c* faults" respectively. If v^- is a summation block, then a derived event is created for each edge incident to v^-, and the backward traversal continues along each incident edge until an input block is visited or until all nodes in G have been visited. When an input block is visited, the variable is added to the tree as a basic event.

Case Study: Armature Dynamics of a Pulse-Width Modulated Solenoid Valve

The case study is a pulse-width modulated solenoid valve. In particular, we focus on modeling the electrical circuit and armature motion. The system model is shown in Figure 5.

Nonlinear differential equations govern the dynamics of the solenoid valve. In particular, a simplified description of the armature dynamics is given in equation 4.

$$mx'' = F_{sol} + A_o P_s - K_s x - C_v x' \quad (4)$$
$$F_{sol} = 0.5 B^2 A/\mu_0 \quad (5)$$
$$B = \varphi/A \quad (6)$$
$$\varphi' = 1/N (v_{sol} - i_{sol} R) \quad (7)$$

The armature is subject to several forces that come from different physical domains, the mechanical, magnetic, and hydraulic ones. The net force on the armature is equal to mx''. The hydraulic force is the product between the orifice area, A_o, and the supply pressure, P_s. The spring force acting on the subsystem is given by the spring constant K_s times the armature displacement x, and the damping coefficient C_v times the armature velocity x'. The solenoid force is given by, F_{sol}. F_{sol} can be further defined by equation 5, where the flux density, B, is a function of the flux, φ, and area of the air gap, A, between the steel armature and the inductor of the electrical circuit. The electrical circuit is described by equation 7, where v_{sol} is the input voltage to the circuit, i_{sol} is the current, R is the resistance, and *N* is the number of loops in the inductor of the circuit. Many more parameters influence the system design, but to reduce space, we only highlight the main equations and their parameters. The parameters are lumped into coefficient blocks in the system model shown in Figure 5. These parameters are: $G_1 = f\,(C_v, m)$, $G_2 = f\,(K_s, m)$, $G_3 = f\,(A_o, m)$, $G_4 = f\,(m)$, $G_5 = f\,(A, \mu_0)$, $G_7 = f\,(R, N)$, $G_8 = f\,(A)$, and $G_9 = f\,(N)$.

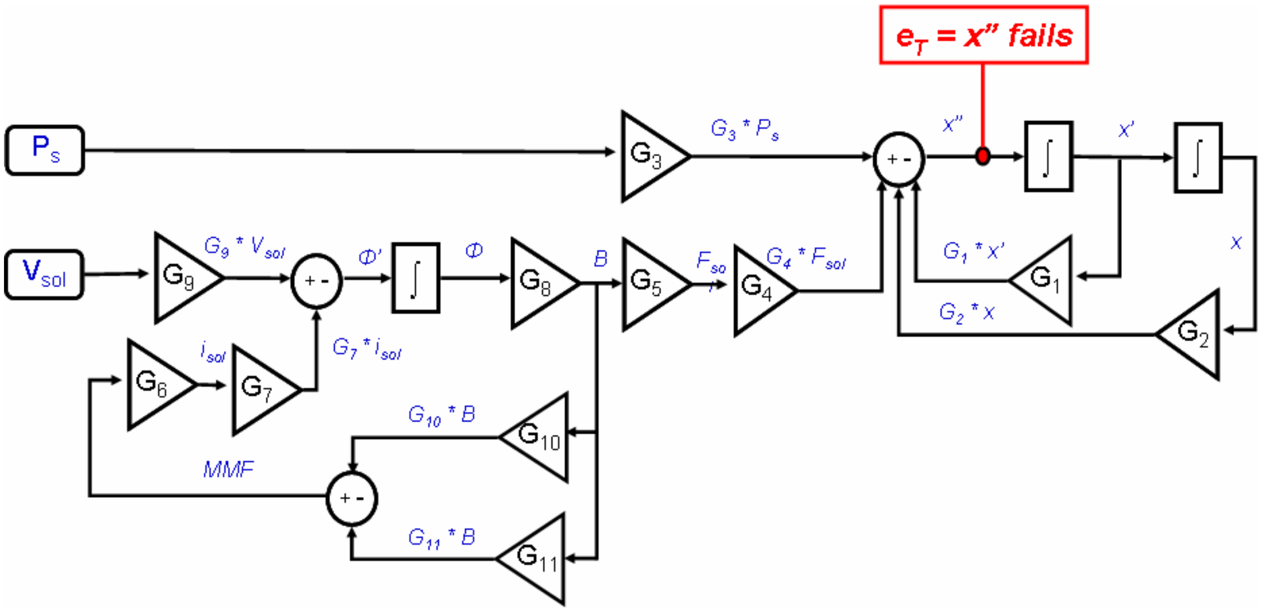


Figure 5 – A Signal Flow Graph of Armature Dynamics

<u>Results</u>: The procedure we used for systematically generating a fault tree is applied to the system model in Figure 5. The resulting fault tree is given in Figure 6. The tree was developed manually, and it took 45 minutes to construct the tree from our procedure. In contrast to practical methods, it only took a single iteration to design the tree because we followed a structured system model that captures the variables and parameters of interest. The fault tree that is produced by our method results in a hierarchical tree with fault events that may be refined to expose additional model parameters and variables.

Looking at the resulting fault tree, one can clearly see the dependency relationship between variables, parameters, and their effects on the top event. The tree allows not only to determine which basic events may lead to the top event, but also how the fault propagates through the system. Input blocks and parameters are basic events since no additional information is given regarding their values or mathematical composition. Variables are derived events produced by input blocks. If numerical values for variables are calculated, it is then possible to define each fault event more precisely, such as “a variable-parameter product is outside its nominal range”, where a nominal range may contain a maximum and minimum real number.

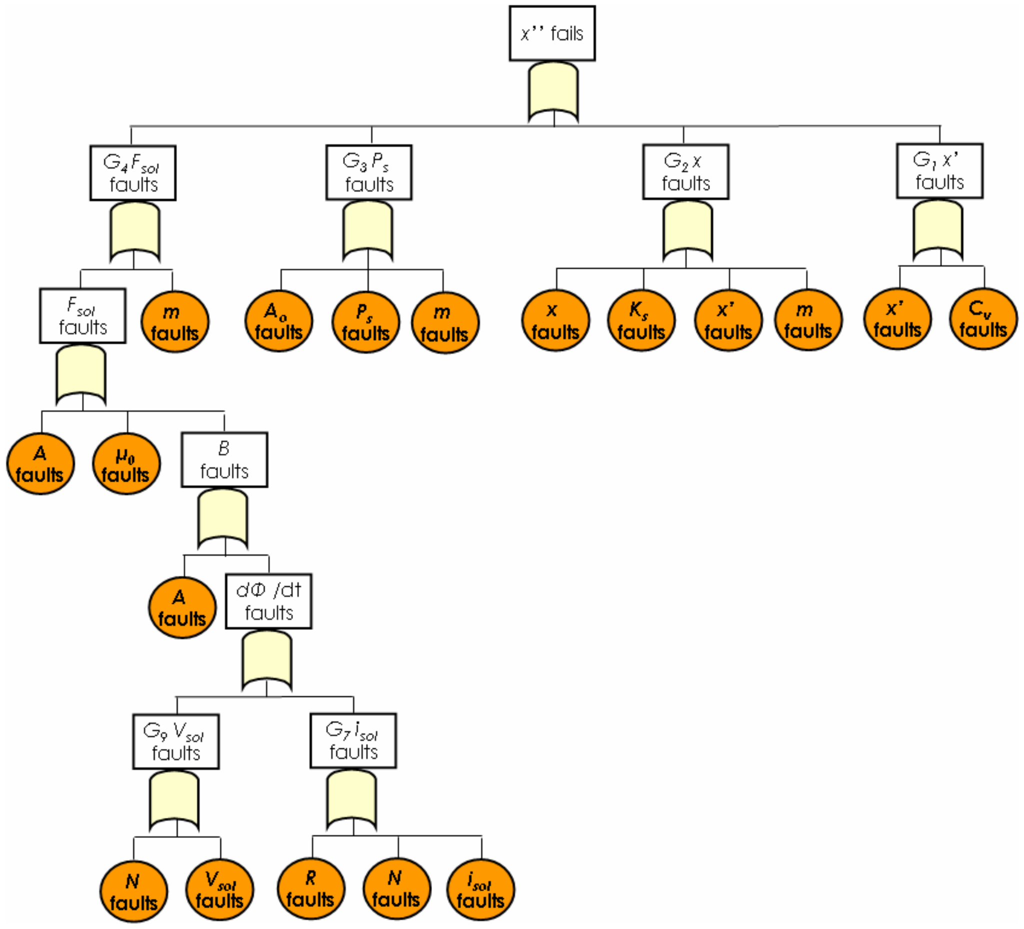


Figure 6 – Generated Fault Tree

Conclusions

We presented a method for generating fault trees systematically that extends our previous work on the same topic. Since the fault tree encodes the cause-and-effect relationship between components, the method presented provides an intuitive way of interpreting fault events at varying levels of abstraction. The system model, which may be described by a set of differential equations, is given as a signal flow graph. The graph may be integrated with our previous work to analyze the dependencies between subsystems.

With the results presented here, we are able to generate fault trees for dynamical systems with a higher level of confidence. The method is systematic, and as a result, it provides structure to intuitive methods of generating fault trees. This also means that the procedure may be automated, much like our previous work in reference 15.

Future Work

As we detailed in Section "Fault Trees and System Models", one must define what a "fault event" is in the given context. We identified two main levels of detail that can be represented in the model and in the corresponding fault tree. Clearly, other levels are also possible, depending on the amount of information, the specific application, or the designers' judgment.

In the *first level*, the object of this paper, the only information given to us is a topological system model. A fault event simply indicates that for a given variable or parameter a potential fault may occur. We do not specify why the fault occurs, nor do we quantify the deviation on the state or on the value of parameters. We just know that a fault "may" occur.

If the user provides more information in the model, then we can yield more details in the fault tree. In a *second level* of detail the user may specify redundancy explicitly, for example inserting objects similar to multiplexers in the data flow. Once the traversal algorithm visits the multiplexer-like object, it will know that only one of the input arcs will be used to produce the output, hence they are redundant. In reference 15, the designers use special FTDF actors and communication media to specify similar redundancy information. Fault events are still defined as the potential for a variable or parameter to fail. As in our previous work (ref. 15), *AND* gates will appear because of redundancy. We are working on an extension of this paper to address the second level of detail.

Acknowledgements

The authors wish to acknowledge constructive discussions with Paolo Giusto. This work is sponsored in part by the Intel Foundation Ph. D. Fellowship, HRL Laboratories L.L.C., General Motors, and the Center for Hybrid Embedded Software and Systems under the NSF ITR CCR-0225610.

References

1. 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.
2. S. A. Lapp and G. J. Powers. "Computer-aided synthesis of fault trees." In IEEE Trans. on Reliability, Vol. R-26, pp. 2-12, 1977.
3. A. Bossche. "Computer-aided fault tree synthesis I (system modeling and causal trees)." In Reliability Engineering and System Safety, Vol. 32, No. 3, pp. 217-241, 1991.
4. S. Ju, C. Chen, and C. Chang. "Constructing Fault Trees for Advanced Process Control Systems—Application to Cascade Control Loops." In IEEE Trans. on Reliability, Vol. 53, No. 1, 2004.
5. J. B. Fussell. "Synthetic tree model: A formal methodology for fault tree construction." ANCR1098, 1973.
6. R. C. De Vries. "An automated methodology for generating a fault tree. In IEEE Trans. on Reliability, Vol. 39, No. 1, 1990.
7. K. K. Vemuri, J. B. Dugan, and K. J. Sullivan. "A design language for automatic synthesis of fault trees." In Procs. of the Annual Reliability and Maintainability Symposium, pp. 91-96, 1999.
8. Y. Papadopoulos and M. Maruhn. "Model-based automated synthesis of fault trees from Matlab-Simulink models." In Procs. of the Intl. Conference on Dependable Systems and Networks, pp. 77-82, July 2001.
9. S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. "Design of embedded systems: Formal models, validation, and synthesis." In Procs. of the IEEE, Vol. 85, No. 3, pp. 366-390, March 1997.
10. The MathWorks. "Stateflow". http://www.mathworks.com, 2006.
11. Modelica Association. "Modelica: A unified object-oriented language for physical systems modeling." http://www.modelica.org, 2005.
12. C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. "Ptolemy II: heterogeneous concurrent modeling and design in Java." Technical Report UCB/ERL M05/21, University of California, Berkeley, 2005.
13. The Metropolis Project Team. "The Metropolis meta model version 0.4." Technical Report UCB/ERL M04/38, 2004.
14. 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, 2004.
15. M. McKelvin, G. Eirea, C. Pinello, S. Kanajan, and A. L. Sangiovanni-Vincentelli. "A formal approach to fault tree synthesis for the analysis of distributed fault tolerant systems." In Procs. of EMSOFT, pp. 237-246, 2005.
16. J. L. Shearer, B. T. Kulakowski, and J. F. Gardner. *Dynamic Modeling and Control of Engineering Systems, 2nd ed*. New Jersey: Prentice-Hall, 1997.
17. D. C. Karnopp, D. L. Margolis, and R. C. Rosenberg. System Dynamics: Modeling and Simulation of Mechatronic Systems, 3rd ed. Wiley and Sons, 2000.
18. H. Tilmans. "Equivalent circuit representation of electromechanical transducers: I. Lumped-parameter systems." Technical Report, Katholieke Universiteit Leuven, 1995.

Biography

Mark L. McKelvin, Jr., Graduate Student, 207 Cory Hall, University of California, Berkeley, CA, 94720, USA, phone – (510) 642-0428, email – mckelvin@eecs.berkeley.edu.

Mr. McKelvin is a Ph. D. student at the University of California, Berkeley (UCB). His research interests include electronic design automation, model-based design of embedded systems for design space exploration, and computer networks. He holds a Master of Science in Electrical Engineering from UCB and a Bachelor of Science from Clark Atlanta University, Atlanta, Georgia. He is a member of Eta Kappa Nu and the National Society of Black Engineers.

Claudio Pinello, General Motors, 2527 Camino Ramon, Suite 360, San Ramon, CA 94583, USA, phone – (510) 378-5240, email – claudio.pinello@gm.com.

Dr. Pinello is a Sr. Research Engineer in the General Motors R&D Organization. He holds an EECS PhD degree from the University of California at Berkeley on design automation for safety critical distributed applications, and a MS degree in Control Theory from the same University on hybrid systems control.

Sri Kanajan, 29096 Gloede Drive, Apt. 8, Warren, MI, 48088, USA, phone – (586) 947-0183, email – sri.kanajan@gm.com.

Mr. Kanajan is a senior research engineer in the Electrical and Controls Integration lab within the General Motors R&D organization. His area of research revolves around electrical architecture design and exploration.

Joseph Wysocki, P.O. Box 2803, Malibu, CA 90265, USA, phone – (310) 612-4264, email – optiquatics@aol.com.

Mr. Wysocki was the manager of the Manufacturing Methods and Materials Department in the Sensors and Materials Lab at HRL Laboratories LLC until his retirement in 2005. His experience spans optical fiber and electronic components reliability, manufacturing processes optimization, design to cost, mission- and safety-critical systems analysis, and most recently, safety-critical by-wire systems.

Alberto Sangiovanni-Vincentelli, Cory Hall, University of California, Berkeley, CA, 94720, USA, phone – (510) 642-4882, email – alberto@eecs.berkeley.edu.

Professor Sangiovanni-Vincentelli holds the Buttner Endowed Chair of EECS at UC Berkeley. He was a co-founder of Cadence and Synopsys. He is a member of the General Motors Scientific and Technology Advisory Board. He received the Distinguished Teaching Award of UC, the worldwide 1995 Graduate Teaching Award of the IEEE for "inspirational teaching of graduate students", and the Kaufman Award of the EDA Council for pioneering contributions to EDA. He authored over 700 papers and 15 books in the area of EDA, embedded and hybrid systems. He is a Fellow of IEEE and a Member of NAE.