OBJECT-ORIENTED FAULT TREE EVALUATION PROGRAM FOR QUANTITATIVE ANALYSES

F. A. Patterson-Hine
NASA Ames Research Center
MS 244-4
Moffett Field, CA 94035

and

B. V. Koen
The University of Texas at Austin
Dept. of Mechanical Engineering
ETC 5.134
Austin, TX 78712

Abstract

Object-oriented programming can be combined with fault tree techniques to give a significantly improved environment for evaluating the safety and reliability of large complex systems for space missions. Deep knowledge about system components and interactions, available from reliability studies and other sources, can be described using objects that make up a knowledge base. This knowledge base can be interrogated throughout the design process, during system testing, and during operation, and can be easily modified to reflect design changes in order to maintain a consistent information source.

An object-oriented environment for reliability assessment has been developed on a Texas Instruments (TI) Explorer LISP workstation. The program, which directly evaluates system fault trees, utilizes the object-oriented extension to LISP called Flavors that is available on the Explorer. The object representation of a fault tree facilitates the storage and retrieval of information associated with each event in the tree, including tree structural information and intermediate results obtained during the tree reduction process. Reliability data associated with each basic event are stored in the fault tree objects. The object-oriented environment on the Explorer also includes a graphical tree editor which was modified to display and edit the fault trees.

The evaluation of the fault tree is performed using a combination of standard fault tree reduction procedures. A bottom-up procedure is used for subtrees that do not contain repeated events, and a top-down, recursive procedure is used to evaluate subtrees that do contain repeated events. The tree is dynamically modularized according to the event which is being evaluated at the time. The locations of repeated events are propagated up the tree and stored in each event object. This information is used to determine which evaluation procedure is required for each event, and intermediate results are stored as they are calculated. Unlike most conventional fault tree evaluation codes which calculate the probability of occurrence of the top event only, this program produces results for every event in the fault tree. The object-oriented approach to fault tree reduction greatly increases the efficiency of the evaluation algorithms.

Introduction

Fault trees are a widely accepted method for modeling complex systems in reliability analyses. A fault tree is a graphical representation of the logical interrelationships among the components of a system [1]. The functional relationships are established in a top-down manner and are represented by logic gates such as AND- and OR-gates. A probability of occurrence is assigned to each base event denoting the failure probability of the components in the system. A fault tree can be evaluated quantitatively to determine the probability of occurrence of the top event in the tree.

The most popular method for quantifying fault trees uses the minimal cut sets of the tree and the component failure information. Minimal cut sets are sets of basic events in which each component in the set must fail for the top event to occur [2]. Algorithms which find the minimal cut sets of a fault tree are better suited to conventional programming languages than are algorithms which directly evaluate the fault tree quantitatively. Quantification of minimal cut sets produces an approximate result in most practical applications, however, since larger trees require the use of truncation techniques to restrict the number of cut sets which are generated.

Object-oriented programming languages enable algorithms which directly evaluate fault trees to be implemented easily and efficiently. These languages provide powerful features which are not available in conventional programming languages. The basic entity in object-oriented programming is the *object*, which is an abstract data type [3]. Objects are defined in terms of *classes* that combine the behavior and state of the objects. Classes describe one or more similar objects, and a particular object is called an *instance* of the class. All data and actions associated with an object are contained in the object's *instance variables* and *methods*. The instance variables represent a private memory for the object, and the procedures, or methods, associated with the object are the only legitimate operators for the data stored in the instance variables. Methods are invoked by sending a *message* to an object, which replaces conventional function calls. A unique feature called *inheritance* allows pieces of code to be shared by several objects. This eliminates unnecessary redundancy in programming and data storage. Instance variables can be accessed via the inheritance hierarchy, so changes that affect the state of all objects of a particular class can be made by updating the class definitions, thus changing at once the information inherited by multiple objects. Inheritance is not provided by any conventional language, and is quite useful in the fault tree application.

The performance of existing codes that evaluate fault trees are limited by the data structures that are available in conventional programming languages. Object-oriented programming provides the flexibility that is needed in defining structures to represent logic gates and basic event data, and the complex interrelationships which exist among them. One of the most serious problems with previous direct evaluation codes is the loss of information about the

system, both pre-defined structural relationships and intermediate results during tree simplification. The application of object-oriented concepts to fault tree analysis results in a clear and concise representation of the tree and provides a powerful mechanism for the storage, retrieval, and evaluation of system information.

Fault Tree Object Definitions

This application has been developed on a Texas Instruments Explorer LISP workstation using the object-oriented extension to LISP called Flavors [4]. Logic gates and basic events share many characteristics which can be described by a general flavor, called TREE. Instance variables that contain information common to both basic events, called nodes, and logic gates are defined in this flavor. All gates and basic events have both a name and at least one parent, so two instance variables, :name and :parent, hold this information. Since the top event of the fault tree does not have a parent, the value of its :parent variable is nil. A variable called :unavailability stores probability data for basic events. Results from the simplification of logic gates is stored in :unavailability for gates. Another variable, :dependent, indicates whether repeated events are located under a particular gate. Basic events are terminal leaves, making the value of the :dependent variable nil for all basic events. The TREE flavor is specialized into two other flavors, GATE and NODE, which contain information particular to logic gates and basic events, repectively.

The GATE flavor includes instance variables that describe more specifically the state of logic gates. All gates are non-terminal leaves, and the names of their children are stored in a variable called :children. A second variable, :dependent-eval, indicates whether or not the gate must be evaluated using the top-down algorithm capable of handling repeated events. The NODE flavor inherits the :unavailability variable from the TREE flavor, but may change the value of :unavailability for all nodes by using a default specification. The :unavailability of each node can also, of course, be changed individually. Specific logic gate types, such as AND- and OR-gates, are described by flavors that are specializations of the GATE flavor. These flavors are necessary for defining methods that apply the specific reduction equations required by each type of gate.

Several other instance variables are defined in these flavors to enable the fault tree to be displayed graphically. The GATE flavor stores a special traversal of all events below and including the particular gate in a variable called :display-trav to be used by the graphical tree editor. NODEs, AND-GATEs, and OR-GATEs include a variable called :flavor-type that contains a description of the defining flavor that is displayed by the tree editor with other information about the events. The TREE flavor stores the name of each object in :name, which is also displayed. The TREE flavor has a component flavor, GWIN:BASIC-GRAPHICS-MIXIN, that furnishes all the information needed for the creation of the graphics objects that describe the fault tree. The tree editor

will be described in the next sections along with the tree reduction algorithms. Figure 1 contains the actual fault tree flavor definitions written in LISP.

```
(DEFFLAVOR TREE
           ((parent nil)
            (name nil)
            (dependent nil))
           (gwin:basic-graphics-mixin)
    :settable-instance-variables)

(DEFFLAVOR GATE
           ((children nil)
            (unavailability nil)
            (dependent-eval nil)
            (display-trav nil))
           (tree)
    :settable-instance-variables)

(DEFFLAVOR AND-GATE
           ((flavor-type 'AND-GATE))
           (gate)
    :settable-instance-variables)

(DEFFLAVOR OR-GATE
           ((flavor-type 'OR-GATE))
           (gate)
    :settable-instance-variables)

(DEFFLAVOR NODE
           ((flavor-type 'NODE)
            (unavailability *default-unavailability*))
           (tree)
    :settable-instance-variables)
```

Figure 1. Fault tree flavor definitions.

Fault Tree Reduction Techniques

The evaluation of the fault tree is performed using a combination of standard fault tree reduction procedures. A bottom-up procedure is used for subtrees that do not contain repeated events, and a top-down, recursive procedure is used to evaluate subtrees that do contain repeated events. The tree is dynamically modularized according to the event which is being evaluated at the time. The locations of repeated events are propagated up the tree and stored in each event object. This information is used to determine

which evaluation procedure is required for each event, and intermediate results are stored as they are calculated. These algorithms are described in detail in [5]. The object-oriented approach to fault tree reduction greatly increases the efficiency of the evaluation algorithms.

Graphical Fault Tree Editor

A general purpose graphical tree editor program that is available in Explorer System 3.0 was adapted to display the fault tree objects. The graphical display of the fault tree enables a visual check of the input tree structure. The user-interface, which includes menu-driven tree editing and pop-up displays of nodal data, allows the user to quickly and easily check the information stored in each fault tree object. Display of the tree after evaluation incorporates the results of tree reduction and modularization information. Probability values can also be edited with a series of menu-driven operations, and the modified tree can be re-evaluated.

The tree editor displays each fault tree object as a rectangle, with the name of the object in the center of the rectangle. GATE objects are shaded, and NODE objects are transparent. The interconnections of the tree objects are shown as lines connecting each node to its parent(s) and children. Figure 2 is a reproduction of a tree editor display as it appears on the Explorer screen. Features such as *zoom in*, *zoom out*, *move left*, and *move right* for tree positioning are accessed from a menu at the top of the display.

Object information other than the name of the object can be displayed by clicking on the specific object of interest. A pop-up window appears that contains the name of each instance variable associated with the object, and its value. Instance variables that are used by the tree editor only are not displayed. Figure 3 is a reproduction of the display after a GATE object is selected for the additional information. The pop-up window displays the GATE's flavor type, parent(s), children, unavailability, dependent list, and its dependent-eval list. The information displayed for a NODE includes flavor type, parent(s), and unavailability.

The probability value of any node in the fault tree can be changed in the graphical tree editor. By clicking on an object in a specified manner, a pop-up window appears that asks for a new probability value for that object. The value entered is then stored in the :unavailability variable for that object. In this way, errors in the object descriptions can be corrected before the tree is evaluated without having to exit the program and edit the data file. The tree may then be evaluated as usual. Modifications to tree objects are not reflected in the input file; however, they are documented in the output file each time the tree is evaluated.

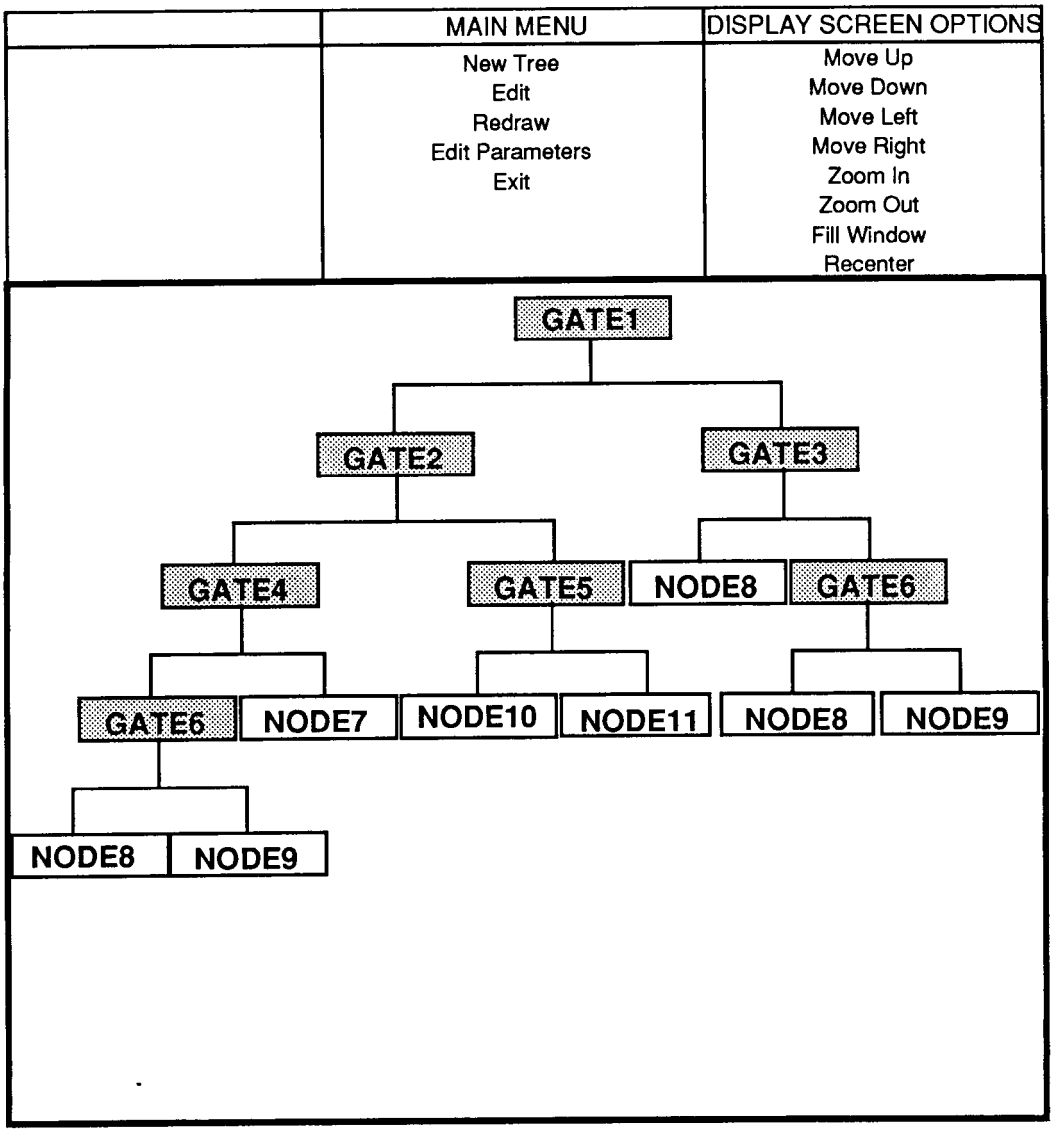

Figure 2 Display of a simple fault tree on the Explorer.

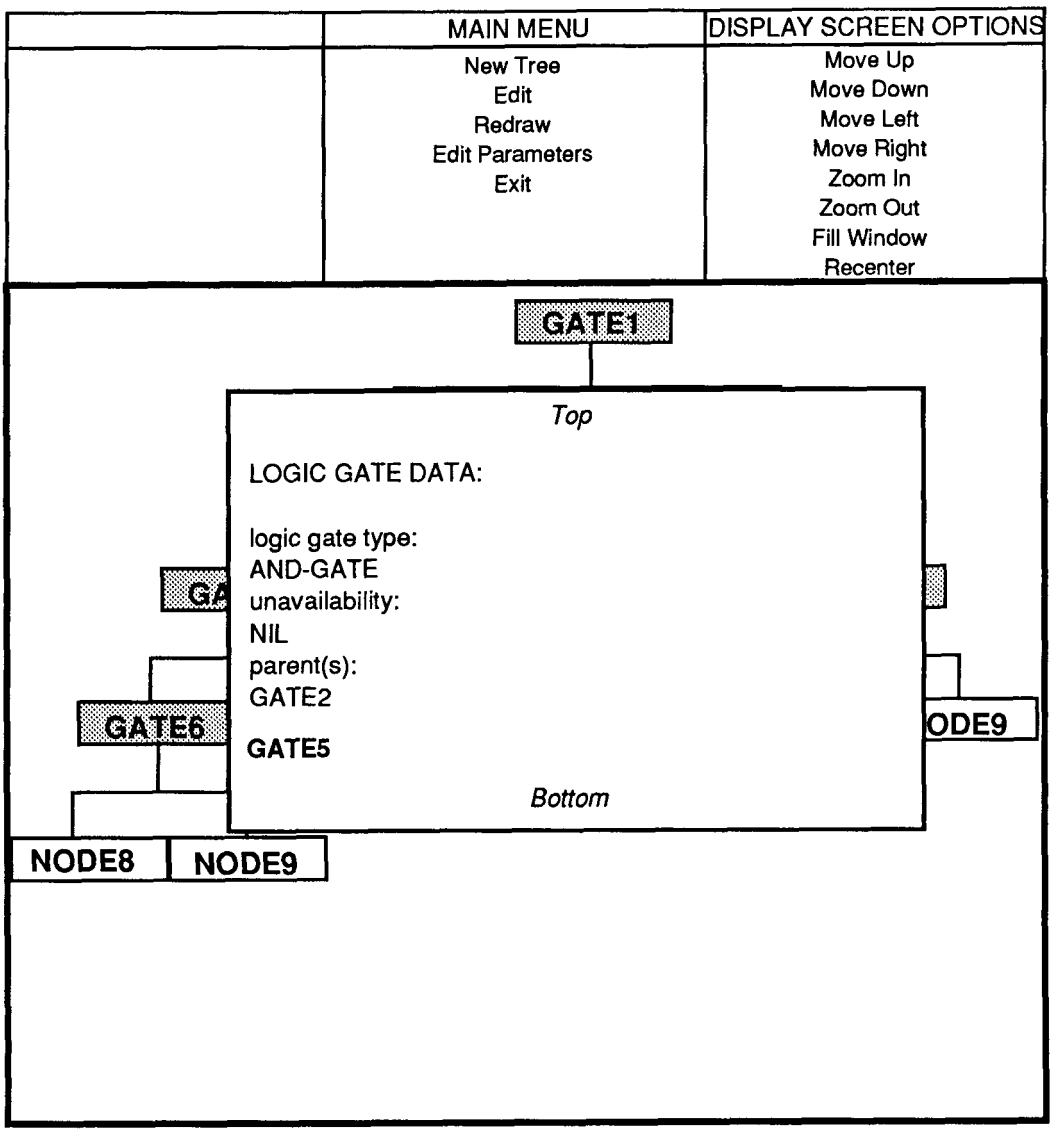

Figure 3 Display of GATE data.

An interesting use of this editing capability is the variation of probability values in order to determine the effects on the probability of occurrence of the top event. For example, a tree can be evaluated and displayed after evaluation. One or more probability values for any tree object, GATE or NODE, can be changed and the tree is then re-evaluated. When such changes are made, the results of the previous calculation are retained for branches that are not affected by the changes. Therefore, calculations are not repeated unnecessarily, and the re-evaluation is completed in as few calculations as possible. None of the most widely used fault tree codes have such editing capabilities.

Conclusions

Object-oriented fault tree techniques provide an improved and flexible environment for reliability analysis. System components are represented by objects which can be organized into a persistent knowledge base of reliability information, improving data consistency. The inheritance heirarchy inherent in the object-oriented environment allows data to be entered either by class for groups of similar components or individually for specific components. Fault trees can be displayed graphically, allowing tree structure to be checked visually. Reliability data for NODEs can also be checked in the tree editor and updated immediately if desired. The tree reduction algorithms perform a direct evaluation of the fault tree and store a probability of occurrence for each event in the event's object. These algorithms are more efficient than previous algorithms implemented in conventional programming languages. The object-oriented environment also enables parameter variation studies to be performed on-line in conjunction with the tree editor.

The flexibility of this environment and the improvements already apparent in the fault tree application suggest that object-oriented fault trees may be appropriate for improving fault detection and diagnosis in complex systems. This topic is currently being explored to provide fault management in the large knowledge-based systems required by space applications.

Acknowledgements

This research was performed while FAPH was a graduate student at the University of Texas at Austin and was sponsored by the NASA Graduate Student Researchers Program, ZONTA International, and NSF grant DMC-8615432.

References:

1. Fussell, J. B., "A Formal Methodology for Fault Tree Construction," *Nuclear Science and Engineeering,* vol. 52, p. 421-432, 1973.
2. Vesely, W. E., Goldberg, F. F., Roberts, N. H., and Haasl, D. F., Fault Tree Handbook, NUREG-0492, 1981.
3. Stefik, M., and Bobrow, D. G., "Object-Oriented Programming: Themes and Variations," *The AI Magazine,* vol. VI, no. 4, p. 40-62, Winter 1985.
4. Explorer Lisp Reference, Texas Instruments, Incorporated, 1987.
5. Patterson-Hine, F. A., Object-Oriented Programming Applied to the Evaluation of Reliability Fault Trees, Ph.D. Dissertation, The University of Texas at Austin, May 1988.