ELSEVIER

0951-8320(95)00050-X

Reliability Engineering and System Safety **49** (1995) 335–353
Elsevier Science Limited
Printed in Northern Ireland.

Demonstration of the Dynamic Flowgraph Methodology using the Titan II Space Launch Vehicle Digital Flight Control System

M. Yau, S. Guarro & G. Apostolakis*

School of Engineering and Applied Science, University of California, Los Angeles, CA 90024-1597, USA

The Dynamic Flowgraph Methodology (DFM) is a new approach for embedded system safety analysis. This methodology integrates the modeling and analysis of the hardware and software components of an embedded system. The objective is to complement the traditional approaches which generally follow the philosophy of separating out the hardware and software portions of the assurance analysis. In this paper, the DFM approach is demonstrated using the Titan II Space Launch Vehicle Digital Flight Control System. The hardware and software portions of this embedded system are modeled in an integrated framework. In addition, the time dependent behavior and the switching logic can be captured by this DFM model. In the modeling process, the dimensionality of the decision tables for software subroutines creates a problem. A possible solution for solving the software portion of the DFM model is suggested. This approach makes use of a well-known numerical method, the Newton-Raphson method, to solve the equations implemented in the subroutines in reverse. Convergence can be achieved in a few steps.

1 INTRODUCTION

This paper discusses the modeling of the Titan II Space Launch Vehicle (SLV) Digital Flight Control Systems (DFCS) using Dynamic Flowgraph Methodology (DFM). The ultimate objective is to analyze the DFM model to identify the major failure modes of the system. The DFM model of the Digital Flight Control System consists of the hardware portion (representing the sensors and the actuators) and the software portion (representing the flight control software). This paper focuses on the discussion of a new approach for solving the software portion of the model. The hardware portion of the model can be solved by the existing DFM solution procedures discussed in earlier DFM publications.

The Titan II SLV Digital Flight Control System can be classified as an embedded system, i.e., a system in which the functions of mechanical and physical devices are controlled and managed by dedicated digital processors and computers. The latter devices, in turn, execute software routines (often of considerable complexity) to implement specific control functions and strategies. In the Titan II SLV Digital Flight Control System, the Missile Guidance Computer (MGC), which is the on-board digital processor, monitors and receives inputs from the electromechanical sensors (the accelerometers, the synchros, and the attitude rate sensors). The computer then executes the flight control software to command the mechanical actuators, namely the thrust deflection gimbal actuators. A discussion of the Titan II embedded system is provided in Section 2.

The Titan II SLV DFCS is used as a test case to demonstrate Dynamic Flowgraph Methodology (DFM). DFM is a new approach for embedded system safety analysis. This methodology is capable of integrating the modeling and analysis of the hardware and software components of an embedded system. The approaches that have been proposed and/or developed in the past generally follow the philosophy of separating out the hardware and software portions of the assurance analysis. The hardware reliability and safety analysts evaluate the hardware portion of an embedded system under the artificial assumption of perfect software behavior. The software analysts, on the other hand, usually attempt to verify or test the correctness of the logic implemented and executed by

* To whom correspondence should be addressed. (Present address: Massachusetts Institute of Technology, Room 24-221, Cambridge, MA 02139-4307, USA).

the software against a given set of design specifications, but do not have any means to verify the adequacy of these specifications against unusual circumstances developing on the hardware side of the overall system, including hardware fault scenarios and conditions not explicitly envisioned by the software designer. A discussion of the limitations of these traditional approaches is presented in Section 3.

DFM is developed to model and analyze an embedded system in a 'systems' approach. This methodology combines features of an existing technique, Logic Flowgraph Methodology (LFM), with discrete state transition models. Thus, DFM provides an inductive (i.e. reverse causality backtracking) analysis capability while at the same time provides the ability to keep track of the complex dynamic effects associated with sequential and time dependent software executions and digital control system behavior. The system analyzed by DFM can either be a design concept or an existing embedded system. This paper shows the application of DFM to an existing system with the software available for analysis. An overview of DFM is given in Section 4.

The DFM model of the Titan II SLV DFCS is developed. This model captures the essential functional and time-dependent behavior of the system. The relationships between the various software and hardware variables in the Titan II system are presented and the execution of the flight control software of this system is modeled as a series of discrete state transitions. One of the steps in developing the model is the construction of decision tables that represent functional relationships between software and/or hardware parameters. However, the problem of combinatorial explosion arises in constructing the decision tables for the Titan II software modules and the resulting tables are too big for storage and easy usage. The approaches to modeling the Titan II SLV DFCS, as well as the dimensionality problem encountered with the decision construction, are discussed in Section 5.

It is recognized that sometimes decision tables need not be constructed prior to the analysis. In many cases, the software module algorithms of the embedded system can be solved 'in reverse' in the inductive (backtracking) analysis, instead of looking up entries in the relevant decision tables. These software module algorithms can be in the form of specification equations or the actual implementation code. Thus, the DFM approach can be applied to test either the design of the embedded system or the actual system itself. The solving of the software module algorithms makes use of a well-known numerical method, the Newton-Raphson method, for solving a system of non-linear equations. A discussion of this approach, as well as an example, is presented in Section 6.

2 THE TITAN II SLV DIGITAL FLIGHT CONTROL SYSTEM (DFCS)

Before the discussion of the limitations of traditional reliability and safety analysis approaches with respect to the Titan II SLV DFCS, and how DFM fills in the deficiencies found in these approaches, we should take a moment to review the features of the system that need to be addressed in the analysis.

2.1 The Titan II system

The Titan II Space Launch Vehicle (SLV) is a modified Titan II ICBM, which is a two stage rocket.[1,2] Stage I provides thrust for the first 150 s of flight to propel the vehicle to the upper atmosphere. Stage I then separates from the SLV and Stage II ignites to power the vehicle to the orbital height. After Stage II shuts down, the vehicle relies on minor attitude adjustments to bring itself to the correct orientations for payload release.

The function of the Digital Flight Control System is to stabilize the vehicle from launch through payload separation. Vehicle attitude control is accomplished via thrust vector control (TVC) from liftoff through Stage II shutdown (powered flight) and attitude control thrusters from Stage II shutdown to payload separation (coast flight). The system also establishes the flight path of the vehicle by implementing all steering commands issued by the guidance system. It should be noted that the flight path of the vehicle is not a fixed trajectory, but only an optimal path that compromises between excessive gravity loss and overheating.

The Digital Flight Control System consists of:
—the Missile Guidance Computer (MGC) and the Flight Control Software;
—the Attitude Rate Sensing System;
—the Inertial Measurement Unit (IMU),
—hydraulic actuators

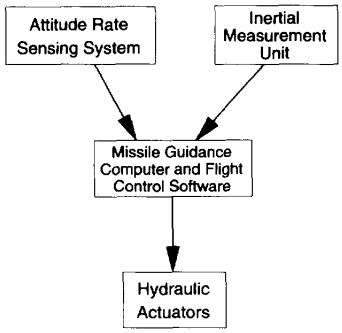

Fig. 1. Block diagram of the Titan II SLV DFCS.

Figure 1 shows a block diagram of the embedded system.

The Inertial Measurement Unit measures the current vehicle orientation and acceleration, while the Attitude Rate Sensing System determines the pitch rate and yaw rate. The measurements from these sensors are then used in the flight control software to determine the appropriate engine nozzle deflection commands to be given to the hydraulic actuators. These sub-systems are each discussed below.

2.1.1 Flight control software

The Titan II flight control software is made up of three major routines; the Real-Time-Interrupt (RTI), the Minor Cycle, and the Major Cycle. Each of these routines is composed of a number of subroutines for carrying out dedicated calculations. Altogether there are more than 50 subroutines, with some of them having more than 200 lines of code written in FORTRAN. In addition, the order of execution of these subroutines is not fixed; there are numerous switching reflecting manoeuvers, shutdown, and changing stages.

The Real-Time-Interrupt is the interface between the software and its environment. It reads in data from the sensing devices, i.e. from the IMU and the rate sensing system, handles telemetry, gives commands to the hydraulic actuators, and issues engine shutdown commands. Inputs are read in and outputs are given out every 40 ms from liftoff to Stage II shutdown (powered flight) or 20 ms after Stage II shutdown (coast flight).

The Minor Cycle makes use of the data read in during the RTI and the guidance information supplied by the Major Cycle to calculate the proper engine deflections. The resulting commands are then issued to the actuators during the Real-Time-Interrupt.

The Major Cycle provides guidance of the flight path. It determines the type of maneuvers to be executed at a certan phase.

The three major routines share the computer resource in the Missile Guidance Computer. The sharing of the computer resources is shown in Fig. 2. The RTI has the highest priority, it is executed every 5 ms and lasts a very short time. All calculations relevant to the Minor Cycle and the Major Cycle are suspended. When it is time to execute the RTI, the computer stores the address of the code it is currently implementing and the results gotten so far. After executing the RTI, the computer resumes the calculation where it left off.

The Minor Cycle has the next highest priority, and completes every 40 ms during powered flight and every 20 ms during coast flight. The actual calculation time is less than 40 ms/20 ms. All the time left during that period will be used up by the Major Cycle calculations.

The Major Cycle has the lowest priority among the three major routines. It completes every 1 s. Calculation takes place in the time-slot unused every Minor Cycle. The major cycle calculation is actually completed in less than 1 s. The time-slot normally used by the Major Cycle after its completion is dedicated to background calculations. Background calculations are not relevant to flight control and will not be discussed here.

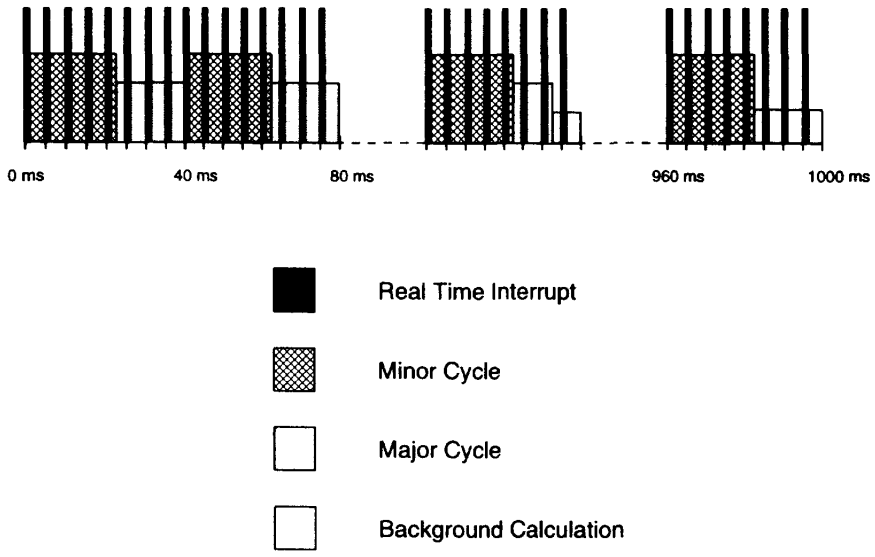

Fig. 2. Sharing of computer resources.

2.1.2 The Attitude Rate Sensing System
The Attitude Rate Sensing System consists of gyros. The gyros measure the pitch rate and the yaw rate of the vehicle. A typical gyro consists of a float spinning with angular momentum **H** in an inertial casing. As input rate ω is sensed, the spinning float will rotate about the output axis according to the physical law $O = \mathbf{H} \times \omega$. Thus, the amount of angular motion can be determined by measuring the rotation about the output axis.

2.1.3 The Inertial Measurement Unit (IMU)
The IMU consists of a platform and associated instruments to measure the vehicle's acceleration and rotations. The acceleration is measured by the accelerometers, while the rotation is measured by the synchros. The accelerometer is made up of a shuttle mass suspended inside a case by a spring. When the case undergoes an acceleration **a**, the shuttle mass moves against the spring due to its inertia. An electromagnet is energized to restore the shuttle mass back to its null position. A measurement of the electromagnet current needed to restore the shuttle mass is then a measurement of **a**.

A synchro is an electromechanical transducer which converts an angular motion θ into a two channel electrical output signal. One channel gives $K \sin(\theta - 120°)$ and the other gives $K \sin(\theta - 60°)$, where K is the instrument's constant.

2.1.4 The hydraulic actuators
The hydraulic actuators are used to deflect the thrust chambers for steering the vehicle. The limit of deflection is approximately 4·93°. There are two thrust chambers in Stage I and one thrust chamber in Stage II.

2.2 Features of the Titan SLV DFCS

As seen from the discussion above, the Titan II SLV Digital Flight Control System is a complex embedded system consisting of numerous hardware components and a large software component. The hardware and software portions of the system are in constant interaction, with the sensors providing readings to the software, and the software giving commands to the actuators. In addition, the system is also dynamic. The software executes in interdependent cycles. The more urgent calculations such as correcting the flight path and determining the thrust deflections are carried out in the minor cycle, while the task of controlling the general flight direction is implemented in the major cycle. Above all, the program is interrupted every 5 ms for reading inputs from the sensors, giving outputs to the actuators, or performing telemetry.

In view of the complexity and dynamic features of this system, in order to analyze the system, we need a methodology that is algorithmic, can handle hardware and software interactions, and can address the dynamic issues. An algorithmic approach can allow its procedures to be automated. The capability for automation is especially important for a huge and complex system like the Titan II SLV DFCS, as the analysis can easily become unmanageable by hand. The last decade has seen much progress in the development and application of analytical techniques to identify possible failure modes in complex engineering systems, and, more recently, even to automatically diagnose faults in real time by means of computer-based operation aids. The recent advent of expert system technology has opened the door to easier implementation of well-known techniques such as fault tree analysis, and to the development of structured knowledge bases on more sophisticated system modeling frameworks such as influence diagrams and qualitative cause-and-effect models.

In addition to following an algorithmic approach, it is obvious that a methodology without due regard to hardware/software interactions and dynamic features cannot be expected to reasonably analyze this system.

3 LIMITATIONS OF TRADITIONAL SOFTWARE RELIABILITY ANALYSIS TECHNIQUES

The traditional software reliability techniques under consideration are testing, formal verification, discrete state simulation, and fault tree analysis. These methodologies are found to have drawbacks when applied to analyzing the Titan II SLV DFCS. The nature and extent of these drawbacks are discussed below.

3.1 Testing

Testing is traditionally one of the most important activities carried out to assure that a given design is, in its actual implementation, complying with certain assigned constraints and specifications, be they in the realm of 'peak performance', safety, or reliability. For systems such as nuclear reactors, aircraft, and spaceships, where failures threaten life, testing costs account for as much as 80% of the total manufacturing cost.[3] This is also true for software systems, where the dominating cost is often not the cost of designing and programming, but the cost associated with logic and implementation errors: the cost of detecting them, the cost of correcting them, the cost of designing tests that discover them, and the cost of running those tests.[3]

In traditional black box testing, the embedded system software is treated as a black box. Combinations of inputs are fed into the software and the

outputs produced are monitored to discover incorrect behavior. The selection of the input domains to be tested is more an art than a science. Choosing the inputs is largely based on judgment. Since system failures usually arise with inputs corresponding to very special circumstances, it is very likely that these inputs will be overlooked in the testing process.

In addition, the amount of sampling involved in black box testing is very large. As the software used in the Titan II SLV Digital Flight Control System is very complex and involves numerous switching actions, a huge set of inputs has to be chosen, and these inputs must cover all the paths reachable via switching actions. The flight control software operates on more than 50 inputs. Assuming we select three values for each input in testing the flight control software, we have to sample more than 3^{50} times (approximately 7×10^{23} times). In reality, we may need more than 3 values for each input to reasonably cover most of the reachable execution paths. A task of this magnitude is nearly impossible and certainly impractical.

The magnitude of sampling inputs can be reduced by performing module testing on the subroutines, interface testing between the subroutines, and then integrating the results. However, this approach still involves executing a huge set of input combinations. For the particular subroutine BLOCK 1, which operates on 9 input variables, selecting 5 values for each input during module testing implies that we still need to sample 5^9 times (almost 2 million times).

Owing to the difficulty in selecting inputs and the magnitude of sampling in black box testing of the whole software or module testing of the subroutines, random testing is impractical and almost impossible when applied to a complex system like the Titan II SLV Digital Flight Control System.

3.2 Formal verification

Formal verification is another approach to software reliability, and is gaining popularity in the software community. Formal verification applies logic and mathematical theorems to prove that certain abstract representations of software, in the form of logic statements and assertions, are consistent with the specifications expressing the desired software behavior. Recent work has been directed at developing varieties of this type of technique specifically for the handling of timing and concurrency problems.[4,5]

Due to the abstract nature of the formalism adopted in formal verification, this approach is rather difficult to use properly by an analyst without a specialized mathematical background. In addition, the complexity in size and functions of the software used in the Titan II SLV DFCS compounds this difficulty. It is not a trivial matter to express this flight control software with more than 10,000 lines of code in terms of abstract logic statements. Finally, formal verification does not provide a framework for modeling and analyzing hardware/software interactions, which is an important issue in analyzing the Titan II SLV Digital Flight Control System as mentioned earlier in Section 2.

3.3 Discrete state simulation

The third type of approach to software assurance is directed at analyzing the timing and logic characteristics of software executions by means of discrete state simulation. In a discrete state simulation, a model is developed to represent the possible paths and states of a software system. The analyst then specifies an initial condition and simulates the behavior of the system in the model. The purpose is to check that the initial condition cannot lead to failure. This approach uses modeling techniques of various types, such as queuing networks and Petri nets.[6–10]

Although this approach can be extended to model combined hardware/software behavior, difficulties arise from the 'march forward' nature (in time and causality) of this type of analysis, which forces the analyst to assume knowledge of the initial conditions from which a system simulation can be started. In a large system such as the Titan II SLV DFCS, many combinations of initial states exist (as in testing) and the solution space easily becomes unmanageable.

3.4 Fault tree analysis

Conventional fault tree analysis is very well established in the areas of safety and reliability analysis. Originally developed at the Bell Laboratories, fault tree analysis has been used to analyze nuclear power plants[11] and chemical processes.[12] Fault tree analysis has also been extended to analyze software systems.[13,14]

The difficulties in applying fault tree analysis to the Titan II SLV Digital Flight Control System can be attributed to the technique's limitations in representing dynamic effects. Fault trees are static diagrams depicting logical combinations of component conditions which lead to a system failure. For the Titan II embedded system, it is an important issue to address how hardware and/or software conditions can evolve over time and lead to system failures. Unfortunately, this issue is not properly addressed by the fault tree analysis approach.

We have seen that the conventional software reliability analysis methodologies are not satisfactory tools for analyzing the Titan II SLV DFCS. These methodologies lack the capability to handle hardware/software interaction, or are limited in dynamic modeling features. A tool which can address the issues identified earlier in Section 2 is needed.

4 OVERVIEW OF THE DYNAMIC FLOWGRAPH METHODOLOGY (DFM)

The Dynamic Flowgraph Methodology (DFM)[15-19] has been developed as a tool to model and analyze embedded systems in a 'systems' approach. Both the hardware and software portions of the embedded system will be represented and analyzed in an integrated framework. It should be observed that the software portion can be in the form of software design specifications or actual software codes. Thus DFM can be applied to validate the system design and catch design errors before it is too late or too expensive to fix them, or to verify the system implementation and check if it matches the design requirements. DFM will be tested using the Titan II SLV DFCS, with the actual software available. Before discussing the DFM approach to the Titan II embedded system in Section 5, this section provides an overview of the methodology itself.

DFM is based on the Logic Flowgraph Methodology (LFM),[20-22] which is a concept for analyzing systems with limited dynamic features. LFM was originally developed as a method for analyzing/diagnosing plant processes with feedback and feedforward control loops. Research conducted over the past decade has demonstrated LFM's usefulness as a tool for computer-automated failure and diagnostic analysis which shows broader potential applicability and efficiency than most other approaches that have been proposed for such objectives.[20] As part of the LFM research effort, models of nuclear power plants and space systems[21-23] have been derived: in addition procedures to be applied in an expert system capable of assisting an analyst in the construction of LFM models have been identified.[24] Recently, the use of LFM and other logic modeling approaches for the safety analysis of aerospace embedded systems has also been investigated.[25,26] The system under consideration in LFM is represented as a logic network relating process parameters at steady state. This LFM model takes the form of a directed graph, with relations of causality and conditional switching actions represented by 'causality edges' and 'conditioning' edges that connect network nodes and special operators. Of these, causality edges represent important process variables and parameters, and conditioning edges represent the different types of possible causal or sequential interactions among them. The LFM models provide, with certain limitations, a complete representation of the way a system of interconnected and interacting components and parameters is supposed to work and how this working order can be compromised by failures and/or abnormal conditions and interactions.

DFM aims at extending the LFM concept to analyzing time-dependent systems. Certain features and rules are added to the LFM to address issues relevant in embedded system analysis not covered by LFM. These issues are:

(1) the need to represent time transitions. Discrete time transitions are almost always present in embedded system software, and often present even in embedded system hardware (e.g., as a result of relay actions);
(2) the need to identify and represent, in a distinguishable manner, the continuous/functional relations and the discontinuous/discrete logic influences that are present in embedded systems.

DFM is a methodology for analyzing and testing embedded system safety. It specifically addresses the fact that embedded system safety analysis cannot be performed effectively if the software and hardware portions of the analysis and qualification process are carried out in relative isolation without a well integrated understanding and representation of the overall system functions and interfaces. Because software testing is expected to remain an important pillar of any dependability verification procedure, this methodology is designed to provide assistance to the software testing process, by helping to identify the most effective test criteria and reduce the amount of 'brute force' testing required for dependability verification. Thus, the ultimate objective is to show how, by using effective analysis techniques, the amount of resources to be committed to the embedded system software verification process could be significantly reduced with respect to what would be required if verification were to be conducted by testing only.

DFM is a tool for analyzing embedded systems with the purpose of (1) identifying how certain postulated events (desirable or undesirable) may occur in a system, and (2) identifying an appropriate testing strategy based on an analysis of the system's behavior. System models which express the logic of the system in terms of causal relationships between physical variables and temporal characteristics of the execution of software modules are analyzed to determine how a certain state (desirable or undesirable) can be reached. This is done by developing timed fault (or success) trees for a given top event (translated in terms of the state(s) of one or more system variables) by backtracking through the model in a systematic manner. These timed fault trees simply take the form of logical combinations of static trees relating the system parameters at different points in time. The resulting information concerning the hardware and software states that can lead to certain events of interest can then be used to identify those software execution paths and associated hardware and environmental conditions which are potentially capable of

leading to serious failures in the implemented system. The verification effort can then be focused specifically on these safety critical features of the system to ensure that such failures cannot be realized.

DFM involves two major steps. In the first step, a model of the embedded system that expresses the logic and dynamic behavior of the embedded system in terms of the hardware and software parameters is constructed. The model integrates together a 'causality network' that describes the functional relationships among hardware and software parameters, a 'conditional network' which represents discrete software behaviors due to conditional switching actions and discontinuous hardware performance due to component failures, and a 'time-transition network' that indicates the sequence in which software subroutines are executed and control actions are carried out.

In the second step, the model developed in the first step is analyzed to determine how the system can reach a certain state (desirable or undesirable). This is done by developing 'timed' fault trees for given top events (translated in terms of the state(s) of one or more hardware/software parameters) by backtracking through the model in a systematic manner. These 'timed' fault trees take the form of a sequence of static trees relating the system parameters at different points in time; essentially a series of snapshots of the system evolution. All the information required to construct 'timed' fault trees is implicitly containd in the DFM model developed in Step 1. In addition, the knowledge base established in Step 1 and the algorithmic approach in backtracking can allow this process to be completely automated. Hence, in Step 2, many 'timed' fault trees can be constructed to analyze different top-events using a single DFM model.

It should be noted that the results of a DFM analysis are obtained in the form of 'timed' fault trees, which show how the investigated system/process states may evolve. The DFM thus shares, in the final form of the results it provides, some of the features of fault tree analysis. The differences, however, are that the DFM approach provides a documented model of the system behavior and interactions, and also a framework to model and analyze time-dependent behavior, both of which fault tree analysis itself does not provide.

4.1 Framework for model construction (Step 1)

As explained above, the first step in DFM is to construct a model of the embedded system. This model is an integration of a 'causality network', a 'conditional network', and a 'time-transition network' which represent the functional behavior, the discontinuous behavior, and the temporal behavior of the embedded system, respectively. The building blocks of these three networks are process variable nodes, transfer boxes, transition boxes, causality edges, and conditioning edges, and they are shown in Fig. 3. These building blocks and the manner in which they are assembled to form the three networks will be discussed in Section 4.1.1 and Section 4.1.2 below.

4.1.1 Building blocks of DFM

4.1.1.1 Process variable nodes The process variable nodes represent physical and software variables that are required to capture the essential functional and/or discrete behavior of the embedded system. The variable represented by a process variable node is discretized into a number of states. The reason for discretization is to simplify the description of the relations between different variables. The choice of the states for a process variable node is often dictated by the logic of the system. For instance, it is natural to set a state boundary at a value that acts as a trigger point for a switching action or a value which indicates the system is progressing towards failure. The number of states for each variable must be chosen on the basis of careful consideration to balance the accuracy of the model with the complexity introduced by higher numbers of variable states.

4.1.1.2 Transfer boxes Transfer boxes link the process variables nodes together to represent relationships between the variables described by these nodes. The way in which these variables vary with each other are described by decision tables associated with a transfer box. The relationship between the input and output process variable nodes is assumed to exist in the same time frame.

A process variable node can be linked to a transfer box via a conditioning edge or a causality edge. A conditioning edge is used when this variable is capable of triggering different switching actions if it takes on

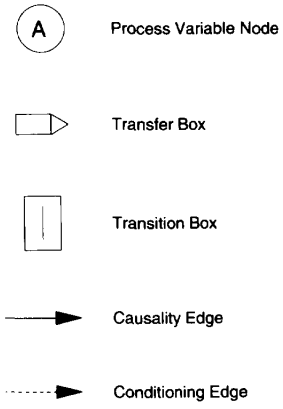

Fig. 3. Building blocks of a DFM model.

different states. On the other hand, a causality edge is used when the variable only exists within the causality flow of the system. The conditioning edge and the causality edge are used to distinguish between functional and discrete behavior found common in embedded systems. The discrete behavior can exist in the form of switching paths in the software or component failures in the hardware.

4.1.1.2.1 Decision tables. A decision table is used to represent the relationships between input and output process variable nodes for a transfer box. This table is a mapping between the combinatorial states of transfer box inputs to the outputs. Decision tables are an extension of truth tables in that they allow each variable to be represented by any number of states.

Decision tables have been used to model components of engineering systems.[27] A system can be modeled in terms of a network of components. Each of these components is characterized by a decision table relating the input states to the output states. This system model, which is made up of components, provides a database for the automatic construction of fault trees. The fault tree construction is implemented by the CAT (Computer Automated Tree) Code.[28]

In a decision table, there will be a column for each input variable and a column for each output variable of interest. The number of rows in the table will equal the product of the number of states into which each input is discretized. If an input process variable node is attached to a transfer box via a conditioning edge, more than one decision table is necessary to map the inputs to the outputs. The number of decision tables is equal to the number of states of this conditioning input.

The information contained in the decision tables is used in building fault trees during the Model Analysis Step (Step 2). In backtracking the DFM model, certain states of a node can be found to cause the top events. The decision table associated with this node is then looked up to find the complete input sets that could have caused those particular states.

4.1.1.3 Transition boxes. Transition boxes are similar to transfer boxes in connecting process variable nodes. Conditioning edges and causality edges are again used to distinguish between discrete and functional behavior. Decision tables are used to describe the relationship between the input nodes and the output nodes. However, transition boxes differ from transfer boxes in the essential aspect that a time lag or time transition is assumed to occur between the time when the input variable states become true and the time when the output variable state(s) associated with the inputs is(are) reached. This time delay is labeled with the transition box.

Transition boxes are used to model portions of an embedded system where timing is critical, such as the execution of software subroutines. In addition, transition boxes can also be used to model hardware time transitions, such as relay actions, which are often found in both embedded systems and conventional control systems.

4.1.1.4 Causality edges. As discussed in Section 4.1.1.2 above, causality edges are used to connect process variable nodes and transfer boxes/transition boxes. The presence of causality edges indicates a cause-and-effect relationship, such as proportional, or inversely proportional between two process variable nodes.

4.1.1.5 Conditioning edges. Unlike causality edges, conditioning edges are used to indicate discrete behavior in the system. They link conditioning parameter nodes to transfer boxes, indicating the possibility of selecting different decision tables in a transfer box. For example, depending on certain software flags, the gains in a controller can be changed, thus altering the functional relationship between the control inputs and control outputs.

4.1.2 Model assembly

To develop a model for an embedded system, physical parameters and software variables that capture the essential causal and temporal behavior of the system are first identified as functional process variable nodes. These nodes are then linked together by transfer boxes and transition boxes via causality edges to form an integrated 'causality' and 'time-transition' network. Discrete behaviors such as component failures and logic switching actions are then identified and represented as nodes linking to transfer/transition boxes via conditioning edges. This 'conditioning network' is then integrated to the 'causality' and 'time-transition' network. The parameters represented by the process variable nodes are discretized into meaningful states dictated by the logic of the system, such as the possibility of triggering switching actions or leading to abnormal behavior. Decision tables are constructed to relate these states together. The completed DFM model then reflects the essential causal, temporal, and logic behavior of the embedded system. The construction of a DFM model is going to be illustrated using the Titan II SLV DFCS in Section 5.

4.2 Framework for model analysis (Step 2)

The second step in DFM is to analyze the embedded system model constructed in Step 1. The result of the analysis is presented as timed fault trees. A timed fault tree takes the form of combinations of conventional 'static' fault trees which describe the

system states at different time steps. It should be noted that the fault trees derived from a DFM analysis are not limited to the binary true/false logic of conventional fault tree analysis. Methods have been developed for finding minimal cut sets (usually called 'prime implicants') of multi-stage logic fault trees.[29,30]

4.2.1 Implementation of Step 2

To construct a timed fault tree, we first have to identify a particular system condition of interest (desirable or undesirable). This system condition is expressed in terms of the states of the process variable nodes. The DFM is then analyzed by backtracking through the network of transfer boxes and transition boxes to find the cause of these variable states. The information discovered at each step of the backtracking process is represented in the timed fault trees.

When analyzing digital control systems with feedback or feedforward characteristics, a node can be traced back to itself in the fault tree construction. Consistency rules must be applied when these situations are encountered. Unlike established consistency checking rules for conventional fault trees which only check against static relationships between variables, these consistency rules must also reflect dynamic relationships; e.g., some variables may not be able to vary independently in time, but must instead satisfy some function of the other variables at different times. This dynamic consistency checking may be performed by developing a rule base, with specific rules for each variable, which reflect that variables allowed dynamic behavior as a function of time and any other necessary variables, as well as constraints on the static relationships between variables.

5 MODELING THE TITAN II SLV DFCS WITH DFM

In constructing the DFM model, a number of simplifying assumptions are made.

(1) The Inertial Measurement Unit (IMU) is assumed to be aligned properly prior to liftoff so that the platform will not drift during flight.
(2) Only the first and last Real-Time-Interrupts (RTI) within a 40 ms time period are represented. This can be justified because readings from sensors and outputs to actuators only occur during these two interrupts. The interrupts in between, occurring every 5 ms, are dedicated to telemetry handling and are not relevant to flight control.
(3) The Major Cycle is represented as one big chunk of calculation at the end of the 1 s period, instead of using up little time-slots left over by the Minor Cycle calculations. This can also be justified because the Minor Cycle calculations only use the Major Cycle results after they are updated.

The hardware and software parameters essential for capturing the behavior of this embedded system are listed in Tables 1 and 2, respectively. For example, the hardware parameters are identified by first recognizing the fact that the function of the embedded system is to stabilize the vehicle during flight, hence B_f, the body axes of the vehicle (roll axis, pitch axis, and yaw axis), is an essential parameter. The body axes can be varied by the deflection of the thrust chambers, so θ_P, θ_R, θ_Y, the deflections of the engines for pitch correction, roll correction, and yaw correction, respectively, are other important parameters. The degree in body axes change is affected by the thrust F, the mass M, the moment of inertia I, and the location of the center of mass CM, and these parameters are added to the list. As the command to engines deflection comes from measuring the gimbal angles α, β, γ, γ_R, the pitch rate PR, the yaw rate YR, and the acceleration along the IMU axes a_{um}, a_{vm}, a_{wm}, these variables are also included in the list. Finally, the IMU measures the gimbal angle by comparing the present body axes and the body axes at go inertial B_i, so the variable B_i is also present in the list.

The DFM model of the embedded system is shown in Fig. 4. Due to the limitation in space, the representation of the flight control software is expanded in detail in Fig. 5. The principal step in integrating the software and hardware portions of the model is the identification of all data that is exchanged between the software and the physical world. In the Titan II system, the software reads in the gimbal angle measurements α, β, γ, and γ_R, and represents them as the variables *H7CH*8, *H7CH*9 *H7CH*10, *H7CH*11, *H7CH*12, *H7CH*13 and *H7CH*16. Similarly, the measurements $\mathbf{a}_{um}$, $\mathbf{a}_{vm}$, and $\mathbf{a}_{wm}$ are converted to accelerometer counts *D5NUC*, *D5NVC*, and *D5NWC*. Finally, the pitch rate and yaw rate are recorded as *W7P1IL* and *W7Y1IL*. On the output side, the D/A outputs *W2DA*11, *W2DA*21, and *W2DA*31 are converted into the thrust chamber deflection angles θ_P, θ_R, and θ_Y.

In Fig. 4, transfer boxes A and F model the IMU, where B_i and B_f are compared to generate the gimbal angle measurements α, β, γ, and γ_R, and the accelerations $\mathbf{a}_{um}$, $\mathbf{a}_{vm}$, $\mathbf{a}_{wm}$ are measured. Transfer box D represents the gyros which measure the pitch rate PR and yaw rate YR. These sensor inputs are used by the flight control software to calculate the thruster deflections θ_P, θ_R, and θ_Y. Finally, transfer box C represents the rocket itself in which the body axes and the current acceleration depends on F, M, I, CM, θ_P, θ_R, and θ_Y.

Figure 5 is the time-transition network constructed based on the control flow of the flight control

Table 1. Hardware parameters of the Titan II DFM model

Variable	Description
$\mathbf{a}_{um}$	Acceleration along the accelerometer *um*-axis
$\mathbf{a}_{vm}$	Acceleration along the accelerometer *vm*-axis
$\mathbf{a}_{wm}$	Acceleration along the accelerometer *wm*-axis
$\mathbf{a}$	Vector acceleration of the vehicle
B_0	Body axes of the vehicle at the beginning of a 40 ms cycle
B_f	Body axes of the vehicle at the end of a 40 ms cycle
B_i	Body axes of the vehicle at go inertial
CM	Center of mass of the vehicle
F	Thrust
I	Moment of inertia about the center of mass
M	Mass of the vehicle
PR	Pitch rate
YR	Yaw rate
α	Platform gimbal angle
β	Middle gimbal angle
γ_R	Outer redundant gimbal angle
γ	Gamma gimbal angle
θ_P	Angle of engine deflection for pitch correction
θ_R	Angle of engine deflection for roll correction
θ_Y	Angle of engine deflection for yaw correction

software. The transition boxes represent software modules, and the nodes represent essential parameters in the software code. For example, RTI-0 is the software module for the first Real-Time-Interrupt which converts the sensor readings into values of software variables. This figure shows how the flight control software calculates the commands to the hydraulic actuators from the sensor measurements.

Note that most of the process parameter nodes are linked to transfer/transition boxes via causality edges. A conditioning edge links the node N8L to the transition box BLOCK 4. N8L is the time kept by the software, and this variable dictates the type of maneuvers to be executed. The execution of different maneuvers causes a discrete jump in the software in the form of using different equations to calculate the desirable body axes.

After linking up the parameters by the transfer boxes and transition boxes, the next step is to construct decision tables to represent the relationships between the parameters. Combinatorial explosion is encountered in the construction of decision tables for the software subroutines. One of the subroutines will be used to illustrate the problem. In the subroutine BLOCK 1, the variables *X*, *Y*, *Z*, *VX*, *VY*, *VZ*, *D7VU*, *D7VV*, and *D7VWA* are used to calculate new values for *X*, *Y*, *Z*, *VX*, *VY*, and *VZ* for the next second. The variables (*X*, *Y*, *Z*) represent the location of the rocket, (*VX*, *VY*, *VZ*) represent the velocity of the rocket, and (*D7VU*, *D7VV*, *D7VWA*) represents the acceleration of the rocket due to thrust alone. These nine variables are each discretized into 5 states, representing a large negative deviation, a moderate deviation, a value close to 0, a moderate positive deviation, and a large positive deviation.

The equations implemented by this software module are listed in Table 3. Equation 1 calculates the acceleration after compensation for the drift of the IMU. Since we assume no drift, the two accelerations are equal. Equations 2–16 transform the accelerations from launch co-ordinates into earth co-ordinates and represent them as *DVSX*, *DVSY*, and *DVSZ*. Equation 17 does nothing more than calculate the square of the magnitude of the acceleration. Equations 18–20 update the position *X*, *Y*, *Z* using the velocities *VX*, *VY*, and *VZ*, and the accelerations *DVSX*, *DVSY*, *DVSZ*, *DVGX*, *DVGY*, and *DVGZ*. The *DVS*'s are accelerations due to thrust alone, while the *DVG*'s are accelerations due to gravitational pull. The total acceleration is a sum of the two. Equations 21–35 update the accelerations due to gravitational pull based on the current position *X*, *Y*, and *Z*. These equations also store the previous accelerations due to gravity as *RE*1, *RE*2, and *RE*3. Finally, eqns 36–38 update the velocities of the vehicle using the accelerations due to thrust and the accelerations due to gravity. The latter is calculated as an average of the current and the previous value.

To complete the entries in the decision table, we need to sample combinations of these 9 input variables. In our model, all these 9 input variables are each discretized into 5 different states. This means that we have to sample at least 5^9 times. Hence, we are running into the combinatorial problem encountered in module testing. In addition, the decision table produced will be huge, consisting of 5^9 rows. It will be very time consuming to look up many tables of this size during the model analysis step. A possible solution to this problem is suggested in the next section.

Table 2. Software parameters for the Titan II Model

D2VUP	Velocity change component in the last 40 ms in launch co-ordinates
D2VVP	Velocity change component in the last 40 ms in launch co-ordinates
D2VWAP	Velocity change component in the last 40 ms in launch co-ordinates
D5NUC	Number of accelerometer counts for the *uc* accelerometer
D5NVC	Number of accelerometer counts for the *vc* accelerometer
D5NWC	Number of accelerometer counts for the *wc* accelerometer
D7VU	Velocity change component in the last 1 s in launch co-ordinates
D7VV	Velocity change component in the last 1 s in launch co-ordinates
D7VWA	Velocity change component in the last 1 s in launch co-ordinates
D8UEL	Numerical derivative of the commanded pitch axis in gimbal co-ordinates
D8UXL	Numerical derivative of the commanded roll axis in gimbal co-ordinates
H7CH8	$K \sin(\alpha - 120°)$
H7CH9	$K \sin(\alpha - 60°)$
H7CH10	$K \sin(\beta - 120°)$
H7CH11	$K \sin(\beta - 60°)$
H7CH12	$K \sin(\gamma_R - 120°)$
H7CH13	$K \sin(\gamma_R - 60°)$
H7CH16	Inner gamma resolver input
I2CHER	Pitch attitude error
R2OLER	Roll attitude error
UET	Guidance desired reference pitch axis in earth co-ordinates
UETC	Guidance desired pitch axis in gimbal co-ordinates
UXI	Guidance desired roll axis in earth co-ordinates
UXIC	Desired roll axis in gimbal co-ordinates
UZEI	Desired yaw axis at initiation of stage I pitchover
U7EDM	Commanded pitch axis in gimbal co-ordinates
U7XDM	Commanded roll axis in gimbal co-ordinates
U8ETA	Commanded pitch axis in gimbal co-ordinates
U8ITA	Commanded roll axis in gimbal co-ordniates
VX	Velocity component in earth co-ordinates
VY	Velocity component in earth co-ordinates
VZ	Velocity component in earth co-ordinates
W2DA11	D/A output
W2DA21	D/A output
W2DA31	D/A output
W2P	Forward loop pitch signal
W2R	Forward loop roll signal
W2Y	Forward loop yaw signal
W2PE10	Pitch attitude error input term
W2RE10	Roll attitude error input term
W2YE10	Yaw attitude error input term
W2PE00	Latest pitch attitude error output term
W2RE00	Latest roll attitude error output term
W2YE00	Latest yaw attitude error output term
W2PLI0	Pitch lateral acceleration input term
W2YLI0	Yaw lateral acceleration input term
W7P1IL	Pitch rate gyro input
W7Y1IL	Yaw rate gyro input
X	Position component in earth co-ordinates
Y	Position component in earth co-ordinates
Y2AWER	Yaw attitude error
Y2DDB	Yaw plane lateral acceleration
Z	Position component in earth co-ordinates
Z2DDB	Pitch plane lateral acceleration

6 A SOLUTION TO THE PROBLEM OF COMBINATORIAL EXPLOSION

This section outlines a new approach to address the combinatorial explosion problem encountered in solving the software portion of the system model. As there is no such problem in the hardware portion of the system, that portion can be analyzed by first constructing decision tables to represent the behavior of the hardware components as in Step 1, and then backtracking these decision tables as in Step 2. Detail discussions on how to carry out these procedures can be found in the earlier DFM references.[15,16] The results obtained from analyzing the hardware portion

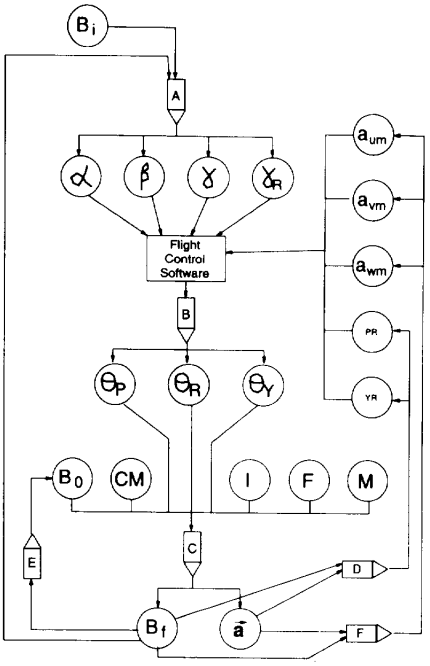

Fig. 4. DFM model of the Titan II flight control system.

of the DFM model are combinations of hardware and software conditions which can lead to system failure. The causes of these software conditions can be identified by using the new solution approach to solve the software portion of the system model.

The new approach to solve the dimensionality problem in the software portion of the DFM model is based on the intention to bypass the construction of decision tables for the software modules. It can be observed that the decision tables constructed in Step 1 are only used for providing information for Step 2 (Model Analysis). These tables provide intermediate information on the backtracking step, namely the parameter states in-between the top events and their causes. If we are able to identify these intermediate causes by some means other than looking up the decision tables, we can avoid the problem altogether.

For the Titan II flight control software, its subroutines implement equations with distinct physical meaning. The equations either represent equations of motion or control laws. Hence, we can take advantage of this fact and try to solve the equations implemented in the subroutines in reverse, instead of constructing and later looking up the decision tables in the backtracking process.

The approach presented here intends to solve the intermediate causes on-line during the analysis. For instance, a particular subroutine is encountered in backtracking the DFM model and certain outputs from this subroutine are found to eventually produce the top event. The next step is to find the combinations of inputs that produce those outputs via this particular subroutine. Instead of looking up the decision tables constructed previously for this subroutine in Step 1, we can try to solve the equations implemented in the subroutine.

For example, the analysts are interested in finding out why the thrust chambers have deflected 2°, 2·5°, and 1·5° respectively for roll correction, pitch correction, and yaw correction. This condition, whose causes are to be found, is represented as the top event in the fault tree shown in Fig. 6. In the DFM model in Figs 4 and 5, we find that the condition $(\theta_P, \theta_R, \theta_Y) = (2°, 2{\cdot}5°, 1{\cdot}5°)$ is backtracked through the transfer box B. This condition is found to be caused by $W2DA11 = 52$, $W2DA21 = 65$, and $W2DA31 = 39$. Next, we backtrack the delay time transition which waits for RTI-7 to execute. The parameters $W2DA11$, $W2DA21$, and $W2DA31$ retain their values. Continuing the backtracking process, the DFM model shows that $W2DA11$, $W2DA21$, and $W2DA31$ are calculated by the subroutine FIG 33 with the input variables $W2P$, $W2R$, and $W2Y$. We need to find what values of these input variables produce the output values of interest in order to enter the gate in the next level in the fault tree. The equations implemented in this software module FIG 33 are solved, and the input values are found to be $W2P = 51{\cdot}7$, $W2R = -12{\cdot}8$, and $W2Y = 51{\cdot}8$. This information is, then, entered into the fault tree. The backtracking process is continued and the equation solving procedure is repeated, if necessary, for the next subroutine. The approach for solving the equations is based on the Newton-Raphson method for solving a system of non-linear equations. An overview of the Newton-Raphson method is provided in Section 6.1, and the discussion of the proposed approach is presented in Section 6.2. An example to demonstrate this approach is provided in Section 6.3.

6.1 The Newton-Raphson method

The Newton-Raphson method[31–33] is a well-known numerical method for solving system of non-linear equations. The Newton-Raphson method can be classified as a fixed-point iteration in which successive guesses are calculated based on previous results to approximate the exact solution. The iteration procedure is terminated when the error becomes less than the predetermined tolerance. The convergence of this solution method is second order, which means that the error in the current iteration is proportional to the square of the error in the previous iteration.

Note that in the iteration, we need to supply an initial guess. This initial guess is crucial to the

Figure 4

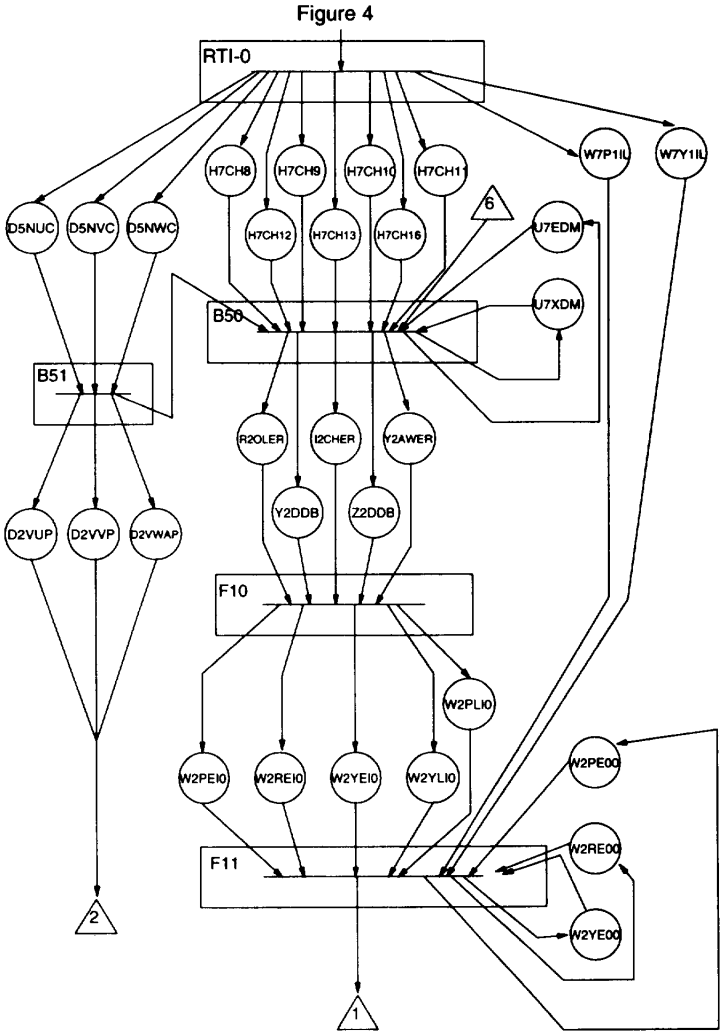

Fig. 5. DFM model of the flight control software.

convergence. If this guess is sufficiently close to the exact solution, the iteration converges rapidly towards the exact solution. On the other hand, if the initial guess is bad, the iteration diverges rapidly. Divergence can also occur if there is no solution to the particular system of equations.

6.2 Solution solving approach

The proposed approach is based on solving the equations implemented by a subroutine in reverse during the backtracking analysis. Owing to the fact that the number of unknowns does not necessarily equal the number of equations, the Newton-Raphson method cannot be applied directly. It has to be adapted to handle the situations in which the number of unknowns is equal to, less than, and greater than the number of equations. These three situations will each be discussed below.

It should be observed that while solving the equations, the analyst must be aware of the switching actions that may exist in the subroutine. This information is identified in Step 1 and is represented in the DFM model as a process variable node linking to the relevant transition box via a conditioning edge. If switching actions arises, the appropriate equations, i.e., the equations relevant to the range of the current iteration, have to be solved.

Suppose a subroutine consists of n input variables $x_1, \ldots, x_n$ and m output variables $y_i, \ldots, y_m$. The

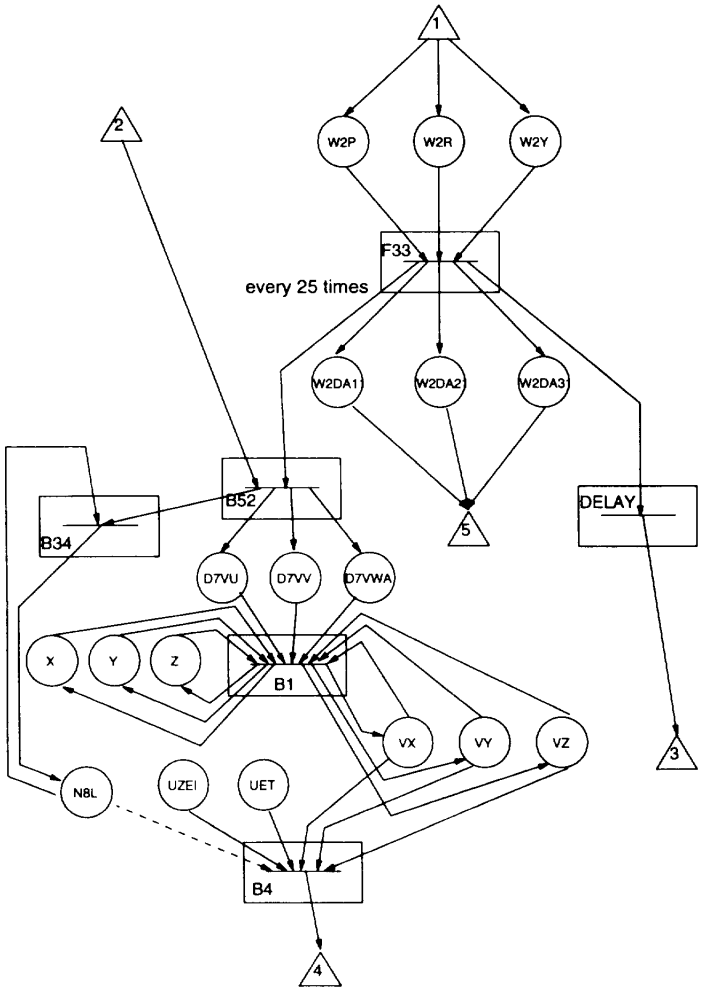

Fig. 5. (cont.)

physical equations implemented by this subroutine are:

$$y_i = g_i(x_1, \ldots, x_n) \quad i = 1, \ldots, m$$

In the backtracking analysis, we discover that the outputs $y_1 = b_1, \ldots, y_m = b_m$ from this subroutine will eventually lead to the top event. The next step is to find out the combinations of input values which produce this set of output values.

We first express the information in terms of m functions:

$$f_i(x_1, \ldots, x_m) = g_i(x_1, \ldots, x_n) - b_i \quad i = 1, \ldots, m$$

The objective is then to determine the values $a_1, \ldots, a_n$ such that

$$f_i(a_1, \ldots, a_n) = 0 \quad i = 1, \ldots, m$$

However, the Newton-Raphson method cannot be applied directly as n does not necessarily equal m. Three different situations may arise when $m = n$, $m > n$, and $m < n$.

Case 1: $m = n$

The Newton-Raphson method can be applied directly. As explained previously, solutions can be found by using a 'good' initial guess for the solution. However, if diverging iterations are produced with this method, it is possible that no solution exists or the initial guess is too far from the exact solution. Further investigations have to be performed to identify which is really the situation encountered.

Case 2: $m > n$

In this case, the output variables $y_1, \ldots, y_m$ are not independent. We can choose amongst them the n

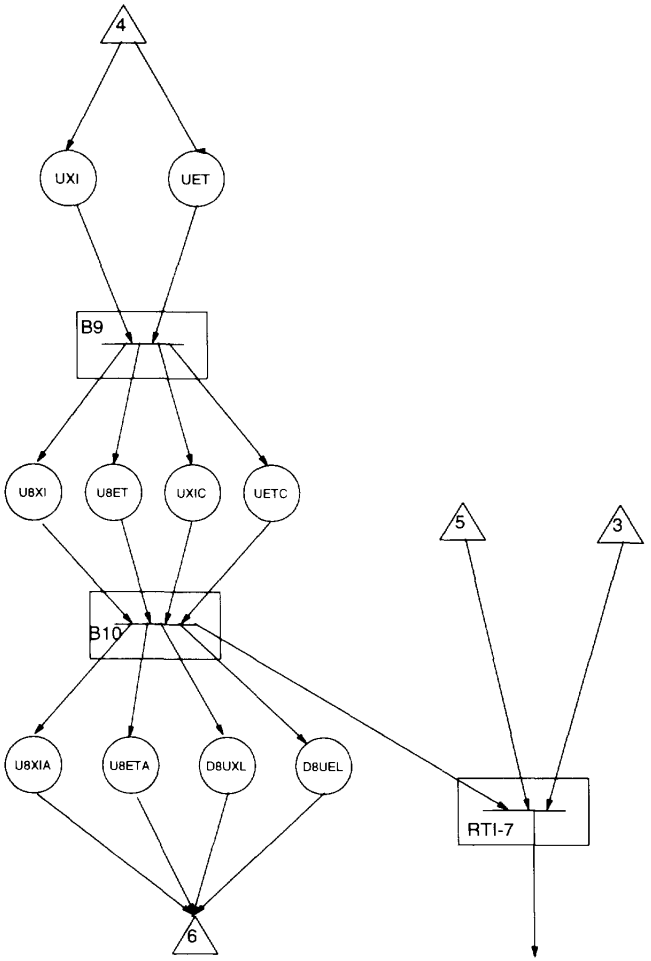

Fig. 5. (cont.)

independent variables. Without loss of generality, let these be $y_1, \ldots, y_n$. We can then solve the system

$$f_i = 0 \quad i = 1, \ldots, n$$

using Newton-Raphson Method, where

$$f_i(x_1, \ldots, x_n) = g_i(x_1, \ldots, x_n) - b_i \quad i = 1, \ldots, n$$

The observation regarding diverging iterations also applies in this situation.

Case 3: $m < n$

The most interesting case occurs when $m < n$. The approach is to assume arbitrary values for $(n - m)$ of the variables and then solve for the remaining m variables. We then repeat this process with another set of arbitrary values for the $(n - m)$ variables.

Suppose we have the subroutine with inputs $x_1, \ldots, x_n$, and outputs $y_1, \ldots, y_m$, and the subroutine implements the equations

$$y_i = g_i(x_1, \ldots, x_n) \quad i = 1, \ldots, m$$

Without loss of generality, we first assume arbitrary values for the last $(n - m)$ variables:

$$x_{m+1} = k_{m+1}, \ldots, x_n = k_n$$

We then solve the system

$$f_i = 0 \quad i = 1, \ldots, m$$

using the Newton-Raphson method, where

$$f_i(x_1, \ldots, x_m) = g_i(x_1, \ldots, x_m, k_{m+1}, \ldots, k_n) - b_i \quad i = 1, \ldots, m$$

Table 3. The subroutine BLOCK 1

Number	Equation
1	$DVW = D7VWA$
2	$FI31 = FI12*FI23 - FI13*FI22$
3	$FI32 = FI13*FI21 - FI11*FI23$
4	$FI33 = FI11*FI22 - FI12*FI21$
5	$RG11 = CG21*FI13 + CG31*FI23 + CG11*FI33$
6	$RG21 = CG22*FI13 + CG32*FI23 + CG12*FI33$
7	$RG31 = CG23*FI13 + CG33*FI23 + CG13*FI33$
8	$RG12 = CG21*FI11 + CG31*FI21 + CG11*FI31$
9	$RG22 = CG22*FI11 + CG32*FI21 + CG12*FI31$
10	$RG32 = CG23*FI11 + CG33*FI21 + CG13*FI31$
11	$RG13 = RG21*RG32 - RG31*RG22$
12	$RG23 = RG31*RG12 - RG11*RG32$
13	$RG33 = RG11*RG22 - RG21*RG12$
14	$DVSX = RG12*D7VU + RG13*D7VV + RG11*DVW$
15	$DVSY = RG22*D7VU + RG23*D7VV + RG21*DVW$
16	$VDSZ = RG32*D7VU + RG33*D7VV + RG31*DVW$
17	$DVSQ = DVSX*DVSX + DVSY + DVSY + DVSZ*DVSZ$
18	$X = X + VX + 1/2*(DVSK + DVGX)$
19	$Y = Y + VY + 1/2*(DVSY + DVGY)$
20	$Z = Z + VZ + 1/2*(DVSZ + DVGZ)$
21	$RSQ = X*X + Y*Y + Z*Z$
22	$R = SQRT(RSQ)$
23	$U = 1/R$
24	$UX = U*X$
25	$UY = U*Y$
26	$UZ = U*Z$
27	$AG = CGMN*U*U$
28	$DVGJ = AG*CJG5*AG$
29	$DVGZE = AG + DVGJ + AG*UZ*DJAG*AG*UZ$
30	$RE1 = DVGZ$
31	$RE2 = DVGY$
32	$RE3 = DVGZ$
33	$DVGX = DVGZE*UX$
34	$DVGY = DVGZE*UY$
35	$DVGZ = (DVGJ + DVGJ + DVGZE)*UZ$
36	$VX = VX + DVSK + 1/2*(DVGX + RE1)$
37	$VY = VY + DVSY + 1/2*(DVGY + RE2)$
38	$VZ = VZ + DVSZ + 1/2*(DVGZ + RE3)$

The detailed procedure for implementing this approach will be demonstrated in Section 6.3.

If no solution exists for the assumed arbitrary values for $x_{m+1}, \ldots, x_n$, the iterations produced in the Newton-Raphson method will diverge no matter what the initial guesses are.

6.3 Example

We will demonstrate this approach using the subroutine BLOCK 1 in the Titan II flight control software. We will see how solving the equations in the software can take the place of looking up big, previously constructed decision tables in the backtracking analysis. This approach is implemented by a software code written in C, and is found to be successful.

The subroutine BLOCK 1 has been presented in Section 5. This subroutine updates the position and velocity of the rocket by using the previously known position and velocity, and the current acceleration of the rocket. The input variables are X, Y, Z, VX, VY, VZ, $D7VU$, $D7VV$, and $D7VWA$. The output variables are X, Y, Z, VX, VY, and VZ.

Suppose in our backtracking analysis, we find that the outputs

$$X = 1{\cdot}001 \times 10^7, \quad Y = 1{\cdot}001 \times 10^7,$$

$$Z = 2{\cdot}002 \times 10^7 \quad VX = -9990,$$

$$VY = 9990, \quad VZ = 19980$$

will eventually produced the top event. This information is found by backtracking the DFM model as seen in Section 6.1. We need to find the inputs in order to go down the next level in the fault tree.

As $n - m = 3$, we first need to assume arbitrary values for three of the input variables. These are

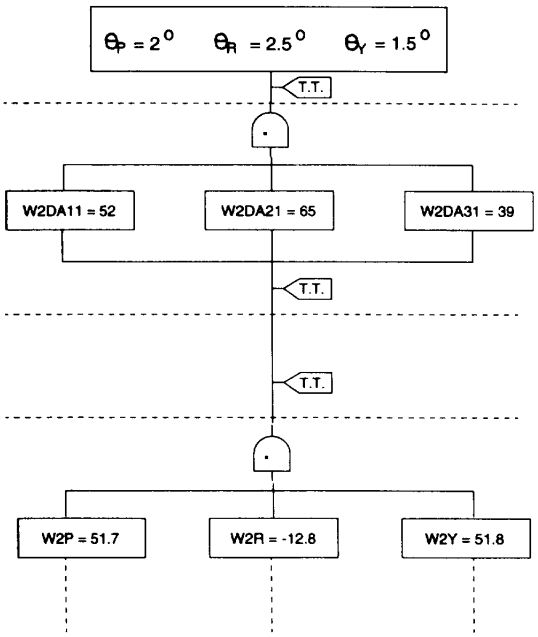

Fig. 6. Fault tree for $(\theta_P, \theta_R, \theta_Y) = (2°, 2{\cdot}5°, 1{\cdot}5°)$.

chosen to be *D7VU*, *D7VV*, and *D7VWA*. The arbitrary values are:

$$D7VU = -100 \text{ or } 100,$$

$$D7VV = -100 \text{ or } 100,$$

$$D7VWA = -100 \quad \text{or} \quad 100$$

The C program tries the eight possible combinations of *D7VU*, *D7VV*, and *D7WA* to solve for *X*, *Y*, *Z*, *VX*, *VY*, *VZ* and uses the initial guesses

$$X_0 = 1{\cdot}001 \times 10^7, \quad Y_0 = 1{\cdot}001 \times 10^7,$$

$$Z_0 = 2{\cdot}002 \times 10^7 \quad VX_0 = -9990,$$

$$VY_0 = 9990, \quad VZ_0 = 19980$$

The results for *D7VU*, *D7VV*, *D7VWA* = 100 Is shown in Table 4. It can be seen that convergence is attained in only two iterations! This is due to the fact that the initial guesses are very close to the exact solution. As a comparison, the program is executed using another set of initial guesses.

$$X_0 = -2{\cdot}1 \times 10^7, \quad Y_0 = 0, \quad Z_0 = 0$$

$$VX_0 = 0, \quad VY_0 = 0, \quad VZ_0 = 0$$

The result produced is shown in Table 5. This time, convergence is attained in three iterations, which is still very fast indeed.

The results shows that solving the equations implemented in the software is a feasible approach to replace the need to construct and look up the decision tables in the backtracking analysis. As seen in Section 5, combinatorial explosion in sampling can be encountered during the decision table construction step. It should be observed that the particular subroutine to be solved and the associated values are indicated by the backtracking of the DFM model. The equations implemented in that subroutine are solved to complete the entries next level down in the fault tree.

7 CONCLUSION

The Dynamic Flowgraph Methodology provides an effective approach for modeling and analyzing an embedded system in a 'systems' framework. This methodology is applicable to very complex systems such as the Titan II embedded system. For the Titan II system, the functional behavior is modeled by process variable nodes, transfer boxes, and causality edges, the discrete behavior is represented by conditioning edges, while the software executions are modeled using the time-transition network. The model developed can be analyzed in Step 2 to identify conditions which cause unwanted behavior and focus the testing effort on these conditions. The algorithmic approach of backtracking and the knowledge base established in Step 1 allows Step 2 to be automated. This is very important for analyzing a complex system such as the Titan II embedded system.

In constructing the decision tables for subroutines in the Titan II flight control software, the problem of dimensionality is encountered. A possible approach for solving the software portion of the DFM model is suggested. This new approach can be applied together

Table 4. Result of iteration

$D7VU = 100, \quad D7VV = 100, \quad D7VWA = 100$

Iteration	*X*	*Y*	*Z*	*VX*	*VW*	*VZ*	Error
0	$1{\cdot}00 \times 10^7$	$1{\cdot}00 \times 10^7$	$2{\cdot}00 \times 10^7$	$-9{\cdot}90 \times 10^3$	$9{\cdot}90 \times 10^3$	$2{\cdot}00 \times 10^4$	$2{\cdot}4 \times 10^4$
1	$1{\cdot}00 \times 10^7$	$1{\cdot}00 \times 10^7$	$2{\cdot}00 \times 10^7$	$-1{\cdot}01 \times 10^4$	$9{\cdot}90 \times 10^3$	$1{\cdot}99 \times 10^4$	$2{\cdot}6 \times 10^{-5}$

Table 5. Result of iteration for a different initial guess
$D7VU = 100$, $DYVV = 100$, $D7VWA = 100$

Iteration	X	Y	Z	VX	VY	VZ	Error
0	$-2{\cdot}0 \times 10^{7}$	0·00	0·00	0·00	0·00	0·00	$3{\cdot}8 \times 10^{7}$
1	$1{\cdot}00 \times 10^{7}$	$1{\cdot}00 \times 10^{7}$	$2{\cdot}01 \times 10^{7}$	$-1{\cdot}01 \times 10^{4}$	$9{\cdot}90 \times 10^{3}$	$1{\cdot}99 \times 10^{4}$	$9{\cdot}7 \times 10^{1}$
2	$1{\cdot}00 \times 10^{7}$	$1{\cdot}00 \times 10^{7}$	$2{\cdot}00 \times 10^{7}$	$-1{\cdot}01 \times 10^{4}$	$9{\cdot}90 \times 10^{3}$	$1{\cdot}99 \times 10^{4}$	9×10^{-10}

with the existing approach for analyzing the hardware portion of the DFM model. Analysis of the hardware portion first identifies combinations of hardware and software conditions which can lead to system failure. The software modules can then be solved in reverse to identify the causes of the software conditions. The solving algorithm is based on the Newton-Raphson Method for solving a system on non-linear equations, with modifications to account for the difference in the number of unknowns and the number of equations. In addition, the model developed in Step 1 will indicate whether switching actions can take place in the particular subroutine. This allows the appropriate equations to be solved. This approach is applied to one of the subroutines in the flight control software. The results demonstrate that convergence can be achieved very rapidly.

ACKNOWLEDGMENTS

The project discussed in this paper was sponsored by the NASA Goddard Space Flight Center.

REFERENCES

1. Martin Marietta Astronautics, *Guidance, Control, and Ground Equations for Flight Plan XX Volume II: Flight Control Equations XX-T001-11-08*, 1988.
2. Martin Marietta Astronautics, *Guidance, Control, and Ground Equations for Flight Plan XX Volume I: Guidance equations XX-U001-1-05*, 1991.
3. Beizer, B., *Software Testing Techniques*, Van Nostrand Reinhold, Scarborough, CA, 1990.
4. Narayana, K. T. & Aby, A. A., Specification of real-time systems in real-time temporal interval logic. In *Proc. 1988 Conf. Real-Time Systems*, IEEE Press, NY, USA, 1988.
5. Razouk, R. R. & Gorlick, M. M., A real-time interval logic for reasoning about executions of real-time programs. In *Proc. ACM SIGSOFT '89, ACM Press Software Engineering Notes*, **14** (1989) 10–19.
6. Dummer, W. G. A., Reiche, H. & Hura, G. S., Petri nets and related graph models. *Microelectronics & Reliab.*, **31** (1991).
7. Morgan, E. T. & Razouk, R. R., Interactive state-space analysis of concurrent systems. *IEEE Trans. on Software Engng*, **SE-13** (1987) 1080–1091.
8. Murata, T., Petri nets: properties, analysis and applications. In *Proc. IEEE*, **77** (1989) 541–580.
9. Leveson, N. G. & Stolzy, J. L., Safety analysis using petri nets. *IEEE Trans. on Software Engng* **SE-13** (1987) 358–363.
10. Peterson, J. L., *Petri Net Theory and the Modeling of Systems*, Prentice-hall, NJ 1981.
11. Henley, E. J. & Kumamoto, H., *Probabilistic Risk Assessment: Reliability Engineering, Design, and Analysis*, IEEE press, NY, USA, 1991.
12. Lapp, S. A. & Powers, G. J., Computer-aided synthesis of fault-trees. *IEEE Trans. on Reliab.*, **R-26** (1977) 2–13.
13. Leveson, N. G. & Harvey, P. R., Analyzing software safety. *IEEE Trans. on Software Engng*, **SE-9** (1983) 569–579.
14. Harvey, P. R., Fault-tree analysis of software. M.S. thesis, University of California, Irvine, 1982.
15. Garrett, C. J., Guarro, S. B. & Apostolakis, G. E., Development of a methodology for assessing the safety of embedded systems. In *2nd Ann. AIAA/USRA/AHS/ASEE/ISPA Aerospace Design Conf.*, Irvine, CA, 16–19 February 1993.
16. Garrett, C. J., Guarro, S. B. & Apostolakis, G. E., The dynamic flowgraph methodology for assessing the dependability of embedded software systems. *IEEE Trans. on Systems, Man, and Cybernetics*, **25** (1995) 824–840.
17. Yau, M., Guarro, S. B. & Apostolakis, G., Demonstration of the dynamic flowgraph methodology using the Titan II Space Launch Vehicle Digital Flight Control System. *UCLA ENG 43-93*, 1993.
18. Garrett, C., Yau, M., Guarro, S. & Apostolakis, G., Assessing the dependability of embedded software systems using the dynamic flowgraph methodology. In *Fourth Int. Working Conf. Dependable Computing for Critical Applications*, San Diego, CA, 4–6 Jan, 1994, pp. 102–119.
19. Garrett, C., Yau, M., Guarro, S. & Apostolakis, G., The dynamic flowgraph methodology: a methodology for assessing embedded system software safety. In *Proc. 2nd Int. Conf. Probabilistic Safety Assessment and Management (PSAM-II)*, San Diego, CA, 20–25 March 1994, pp. 052-1–052-6.
20. Guarro, S. B. & Okrent, D., The logic flowgraph: a new approach to process failure modeling and diagnosis for disturbance analysis applications. *Nucl. Tech.*, **67,** (1984) 348–359.
21. Guarro, S. B., A logic flowgraph based concept for decision support and management of nuclear plant operation. *Reliab. Engng & System Safety*, **22** (1988).
22. Guarro, S. B., Diagnostic models for engineering process management: a critical review of objectives, constraints and applicable tools. *Reliab. Engng & System Safety*, **30** (1990) 21–50.
23. Ting, Y. T. D., Space nuclear reactor system diagnosis: a knowledge based approach, Ph.D. dissertation, UCLA, 1990.
24. Guarro, S. B., PROLGRAF-B: A knowledge-based system for the automated construction of nuclear plant diagnostic models. *Technical Progress Report for Period*

Sept. 1987—March 1988 (by D. Okrent and G. Apostolakis) for DOE Award no. DE-FGO3-UCLA, 1988.
25. Guarro, S. B., Wu, J. S., Apostolakis, G. R. & Yau, M., Embedded system reliability and safety analysis in the UCLA ESSAE project. In *Proc. Int. Conf. Probabilistic Safety Assessment and Management (PSAM)*, Beverly Hills, CA, 4–7 February 1991, pp. 383–388.
26. Muthukumar, C. T., Guarro, S. B. & Apostolakis, G. E., Logic flowgraph methodology: a tool for modeling embedded systems. In *Proc. IEEE/AIAA 10th Digital Avionics Systems Conf.*, Los Angeles, CA., 14–17 October 1991, pp. 103–109.
27. Salem, S. L., Apostolakis, G. E. & Okrent, D., A new methodology for the computer-aided construction of fault trees. *Annals. of Nucl. Energy*, **4** (1977) 417–433.
28. Salem, S. L., Wu, J. S. & Apostolakis, G. E., Decision table development and application to the construction of fault trees. *Nucl. Tech.*, **42** (1977) 51–64.
29. Caldarola, L., Fault tree analysis with multistate components. In *Synthesis and Analysis Methods for Safety and Reliability Studies* (eds G. E. Apostolakis, S. Garribba, G. Volta) Plenum Press, NY, 1980, pp. 199–248.
30. Shields, E. J., Apostolakis, G. E. & Guarro, S. B., Determining the prime implicants for multi-state embedded systems. In *Proc. 2nd Int. Conf. Probabilistic Safety Assessment and Management (PSAM-II)*. San Diego, CA, 20–25 March 1994, pp. 052-7–052-11.
31. Fröberg, C. E., *Numerical Mathematics: Theory and Computer Application*, Benjamin/Cummings, 1985.
32. Johnson, L. W. & Riess, R. D., *Numerical Analysis*, Addison-Wesley, Reading, MA, 1982.
33. Maron, M. J., *Numerical Analysis: A Practical Approach*, MacMillan, NY, USA, 1987.