UCLA ENG 43-93
June 1993

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

M. Yau, S. Guarro, G. Apostolakis

Mechanical, Aerospace and Nuclear Engineering Department
University of California
Los Angeles, CA 90024-1597

Table of Contents

List of Figures

List of Tables

Acknowledgments

The project discussed in this report was sponsored by Grant NAG 5-1440 from the NASA/Goddard Space Flight Center and Grant NCC 2-374 from the NASA Ames Dryden Flight Research Facility. The authors wish to thank Larry Hyatt of the NASA/Goddard Space Flight Center and Victoria Regenie of the NASA Ames Dryden Flight Research Facility for supporting this project.

Abstract

Dynamic Flowgraph Methodology (DFM) is a new approach developed to integrate 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, it is found that constructing decision tables for software subroutines is very time consuming. A possible solution 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 report discusses the modeling of the Titan II Space Launch Vehicle (SLV) Digital Flight Control System (DFCS) using Dynamic Flowgraph Methodology (DFM). The eventual objective is to make use of the DFM model to perform a safety analysis of the system.

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. Further discussion of the Titan II SLV Digital Flight Control System can be found in Section 2 and Appendix C.

The Titan II SLV DFCS is used as a test case to demonstrate Dynamic Flowgraph Methodology (DFM). DFM is a new approach developed to integrate 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 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

X

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 application of DFM to analyze the design of an embedded system with only the software specifications can be found in [Garrett, 1993]. A discussion of LFM is presented in Appendix A. A brief overview of DFM is given in Section 4, and a more in-depth discussion of this approach can be found in Appendix B.

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 combinatorial explosion 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 either test 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 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. A discussion in greater detail about the Titan II SLV DFCS can be found in Appendix C.

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 [Martin Marietta, 1988], [Martin Marietta, 1991]. Stage I provides thrust for the first 150 sec after liftoff to propel the vehicle to the upper atmosphere. After Stage I separates from the SLV, Stage II ignites to power the vehicle to the orbital height. At the end of the Stage II thrust decay, 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 (DFCS) is to stabilize the vehicle during all phases of flight (launch through payload separation). Vehicle attitude control is accomplished via thrust vector control (TVC) during powered flight (from liftoff through Stage II shutdown) and attitude control thrusters during coast flight (from Stage II shutdown to payload separation). 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 fixed. It is only an optimal path that balances 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

Figure 2.1 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. A detailed description of each of these sub-systems is presented in Appendix C.

2.2 Features of the Titan II SLV DFCS

The Titan II SLV Digital Flight Control System is a complex embedded system consisting of numerous hardware components and a large software code component. The software code is made up of more than 50 subroutines, and some of the subroutines have more than 200 lines of code written in FORTRAN. The execution order of the subroutines is not fixed; there are numerous switching reflecting maneuvers, shutdown, and changing stages. In addition, 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.

Besides the complexity in size and functions, the system is also dynamic. The software is executed in interdependent cycles; a minor cycle of 40 ms or 20 ms depending on the flight stage, and a major cycle of 1 s. 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. Additionally, 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, 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

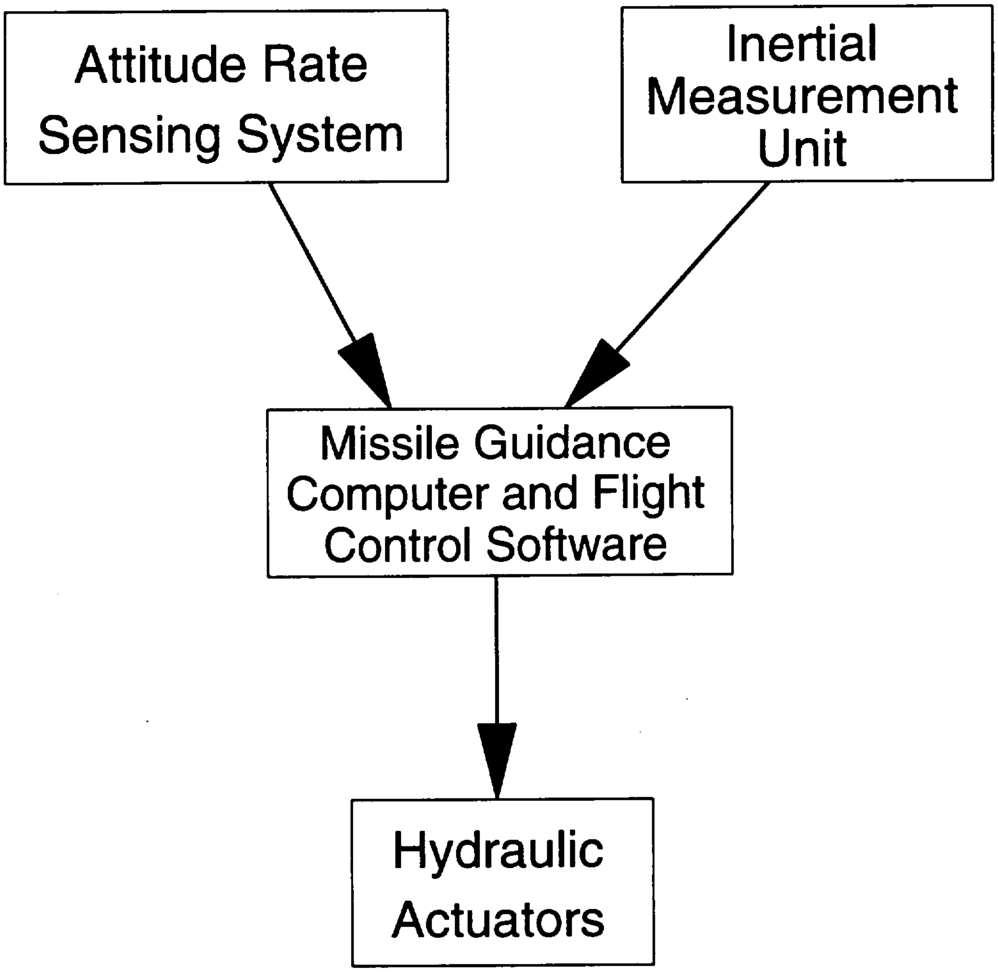

Attitude Rate Sensing System
Inertial Measurement Unit
Missile Guidance Computer and Flight Control Software
Hydraulic Actuators

Figure 2.1 : Block Diagram of the Titan II SLV DFCS

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 extend 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 [Beizer, 1990]. 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 [Beizer, 1990].

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 judgement. 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 3 values for each input in testing the flight control software, we have to sample more than 3^{50} times (approximately 7 x 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 [Narayana and Aby, 1988], [Razouk and Gorlick, 1989].

Due to the abstract nature of the formalism adopted in formal verification, this approach is rather difficult to use properly by an analyst without the 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 [Dummer, Reiche, and Hura, 1991], [Morgan and Razouk, 1987], [Murata, 1989], [Leveson and Stolzy, 1987].

For example, in a Petri net analysis, a model is first developed to describe the possible states of a system, and how the system can change from one state to another. This Petri net model is made up of places, transitions, input functions, and output functions.

The places are used to represent states of the sub-modules making up the system. Hence, a combination of places characterizes a particular state of the system. To help visualize this, tokens can be put into these places. Thus, a particular distribution of tokens in the places represent a particular state of a system.

Transitions link the places together to represent the change of states of the sub-modules. The mapping between the input places and the output places for a transition is described by its input function and output function. A transition is enabled when each of its input places has at least one token. The change of state is modeled by removing the tokens from the input places and depositing them into the output places. Thus, the distribution of the tokens among the places is altered, representing a jump from one system state to another.

Once this Petri net model is established, the analyst can define an initial condition for simulation by specifying a particular distribution of tokens. The enabled transitions are then fired to simulate change of state of the system. The new distribution of the tokens will enable a new set of transitions, and the simulation process is continued. The jump from state to state is summarized in the form of a reachability graph. This graph describes the path of the system transition and its intermediate steps. The reachability graph is then checked to see if a hazardous state (corresponding to a distinct distribution of tokens) can be reached from the particular initial condition. A detailed discussion of Petri net analysis can be found in [Peterson, 1981].

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 [Henley and Kumamoto, 1991] and chemical processes [Lapp and Powers, 1977]. Fault tree analysis has also been extended to analyze software systems [Harvey, 1982], [Leveson and Harvey, 1983].

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) 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 actual software codes or software design specifications. Thus DFM can be applied during the design stage or after the completion of the embedded system. 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 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.

DFM is based on the Logic Flowgraph Methodology (LFM) [Guarro and Okrent, 1984], [Guarro, 1988], [Guarro, 1990], [Muthukumar, Guarro, and Apostolakis, 1991], which is a concept for analyzing systems with limited dynamic features. The system under consideration in LFM is represented as a logic network relating process parameters at steady state. A discussion of the Logical Flowgraph Methodology is presented in Appendix A. Certain features and rules are added to the LFM to address issues relevant in embedded system analysis not cover by LFM. These issues are:

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

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 different software subroutines are executed and different 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 contained in the DFM model developed in Step 1. This backtracking process does not rely on ad-hoc knowledge. 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 different "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. The establishment of a knowledge base, and the capability for automation in this methodology offers great advantages in analyzing complex embedded systems like the Titan II SLV Digital Flight Control System. Once a DFM model has been developed, it is not necessary to prepare separate ad-hoc models for each system state of interest (as it is in fault tree analysis). Also, the dynamic capability of DFM is essential in handling the time-dependent behavior of the flight control system.

Since this paper deals with the modeling of Titan II SLV Digital Flight Control System using DFM, the essential features for Step 1 (Model Construction) are presented. Readers who are interested in Step 2 (Model Analysis) should refer to the discussion in Appendix B.

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 Figure 4.1. 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

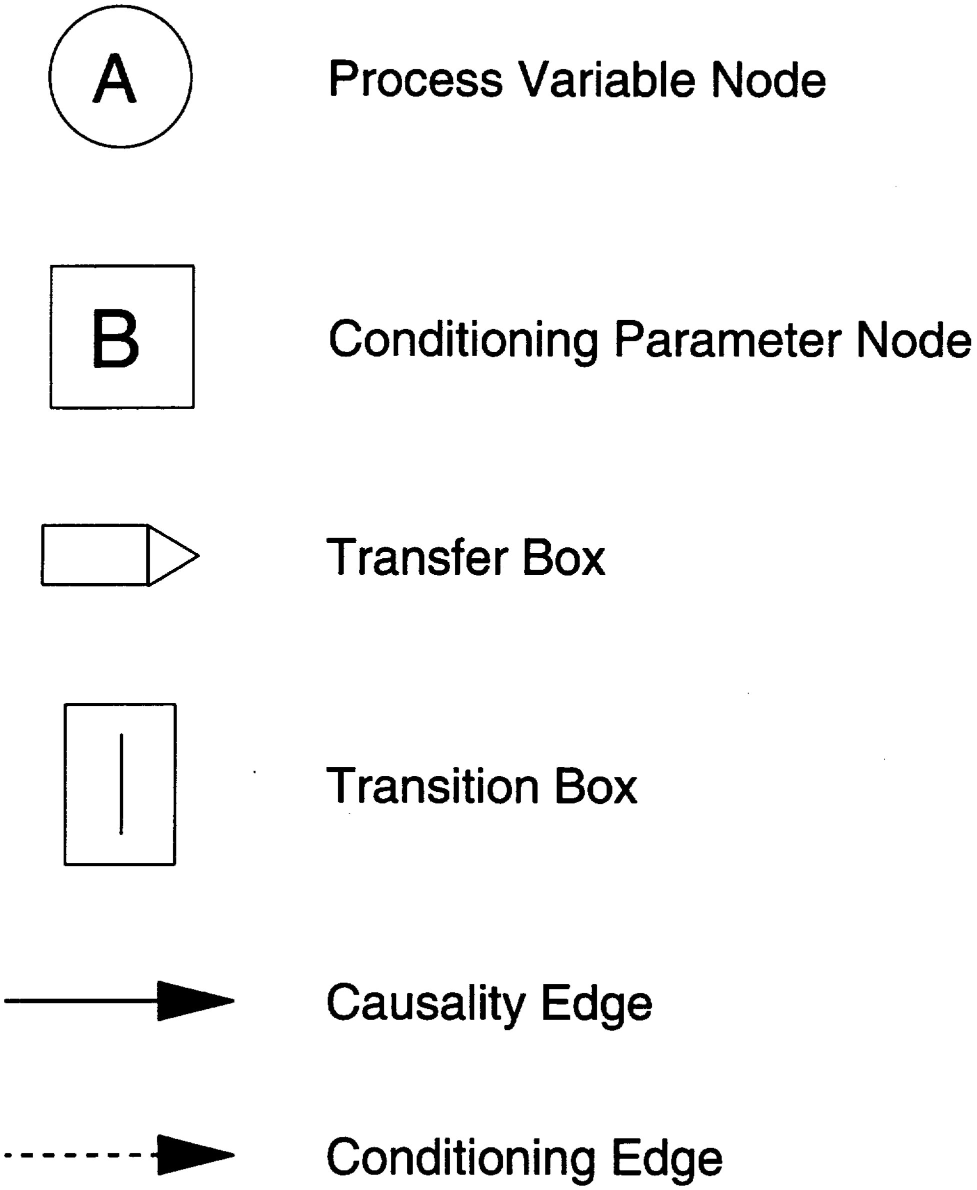


Figure 4.1 : Building Blocks of a DFM Model

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 variable 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 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 [Salem, Apostolakis, Okrent, 1977]. 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 [Salem, Wu, and Apostolakis, 1979].

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.

In the case when an input process variable node is attached to a transfer box via a conditioning edge, more than one decision tables are 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 flag, 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.

5. Modeling the Titan II SLV DFCS with DFM

A DFM model is developed for the Titan II SLV DFCS for Stage I flight. 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 Table 5.I and Table 5.II 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.

a_{um}	Acceleration along the accelerometer um-axis
a_{vm}	Acceleration along the accelerometer vm-axis
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 msec cycle
B_f	Body axes of the vehicle at the end of a 40 msec 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 deflection of the engines for pitch correction
θ_R	Angle of deflection of the engines for roll correction
θ_Y	Angle of deflection of the engines for yaw correction

Table 5.I : Hardware Parameters of the Titan II DFM Model

D2VUP	Velocity Change in the Last 40 msec in Launch Co-ordinates
D2VVP	Velocity Change in the Last 40 msec in Launch Co-ordinates
D2VWAP	Velocity Change in the Last 40 msec 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 in the Last Second in Launch Co-ordinates
D7VV	Velocity Change in the Last Second in Launch Co-ordinates
D7VWA	Velocity Change in the Last Second in Launch Co-ordinates
D8UXL	Numerical Derivative of the Commanded Roll Axis in Gimbal Co-ordinates
D8UEL	Numerical Derivative of the Commanded Pitch Axis in Gimbal Co-ordinates
H7CH8	$K \sin(\alpha-120^{\circ})$
H7CH9	$K \sin(\alpha-60^{\circ})$
H7CH10	$K \sin(\beta-120^{\circ})$
H7CH11	$K \sin(\beta-60^{\circ})$
H7CH12	$K \sin(\gamma_R-120^{\circ})$
H7CH13	$K \sin(\gamma_R-60^{\circ})$
H7CH16	Inner Gamma Resolver Input
I2CHER	Pitch Attitude Error
R2OLER	Roll Attitude Error

Table 5.II : Software Parameters for the Titan II Model (1/3)

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
U8XIA	Commanded Roll Axis in Gimbal Co-ordinates
VX	Vehicle Velocity in Earth Co-ordinates
VY	Vehicle Velocity in Earth Co-ordinates
VZ	Vehicle Velocity 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
W2PEI0	Pitch Attitude Error Input Term
W2REI0	Roll Attitude Error Input Term
W2YEI0	Yaw Attitude Error Input Term

Table 5.II : Software Parameters for the Titan II Model (2/3)

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 1 Gyro Input
W7Y1IL	Yaw Rate 1 Gyro Input
X	Vehicle Position in Earth Co-ordinates
Y	Vehicle Position in Earth Co-ordinates
Y2AWER	Yaw Attitude Error
Y2DDB	Yaw Plane Lateral Acceleration
Z	Vehicle Position in Earth Co-ordinates
Z2DDB	Pitch Plane Lateral Acceleration

Table 5.II : Software Parameters for the Titan II Model (3/3)

The DFM model of the embedded system is shown in Figure 5.1. Due to the limitation in space, the representation of the flight control software is expanded in all the detail in Figure 5.2.

In Figure 5.1, transfer box A models the IMU, where B_i and B_f are compared to generate the gimbal angle measurements α, β, γ, and γ_R. Transfer box D represents the gyros which measure the pitch rate PR and yaw rate YR, while transfer box F shows the accelerometers which provide measurements a_{um}, a_{vm}, a_{wm}. The sensor inputs α, β, γ, γ_R, PR, YR, a_{um}, a_{vm}, and a_{wm} 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 5.2 is constructed based on the control flow in the flight control 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. This module reads in the gimbal angle measurements α, β, γ, and γ_R, and represents them as the variables H7CH8, H7CH9, H7CH10, H7CH11, H7CH12, H7CH13, and H7CH16. Similarly, the measurements a_{um}, a_{vm}, and a_{wm} are read in as accelerometer counts and are represented as D5NUC, D5NVC, and D5NWC. Finally, the pitch rate and yaw rate are represented as W7P1IL, and W7Y1IL. This figure shows that the sensor measurements are read in by RTI-0. The accelerometer counts are first converted to accelerations by the software module BLOCK 51. The next subroutine executed, BLOCK 50, uses the gimbal angle measurements to determine the lateral acceleration of the vehicle and its current body axes. Next, the desirable body axes, which is updated every second in the major cycle, is compared with the current axes to find the attitude errors. The final outputs produced by BLOCK 50 are the attitude errors (roll error, pitch error, and yaw error), and the lateral accelerations (pitch acceleration and yaw acceleration). These variables, together with the pitch rate and yaw rate information, are used by the subroutine FIG 10 to calculate the flight control inputs. These inputs are used in the module FIG 11 to calculate the forward loop signals. The control equations implemented in FIG 11 are 4th order equations, so previous flight control inputs are read in and

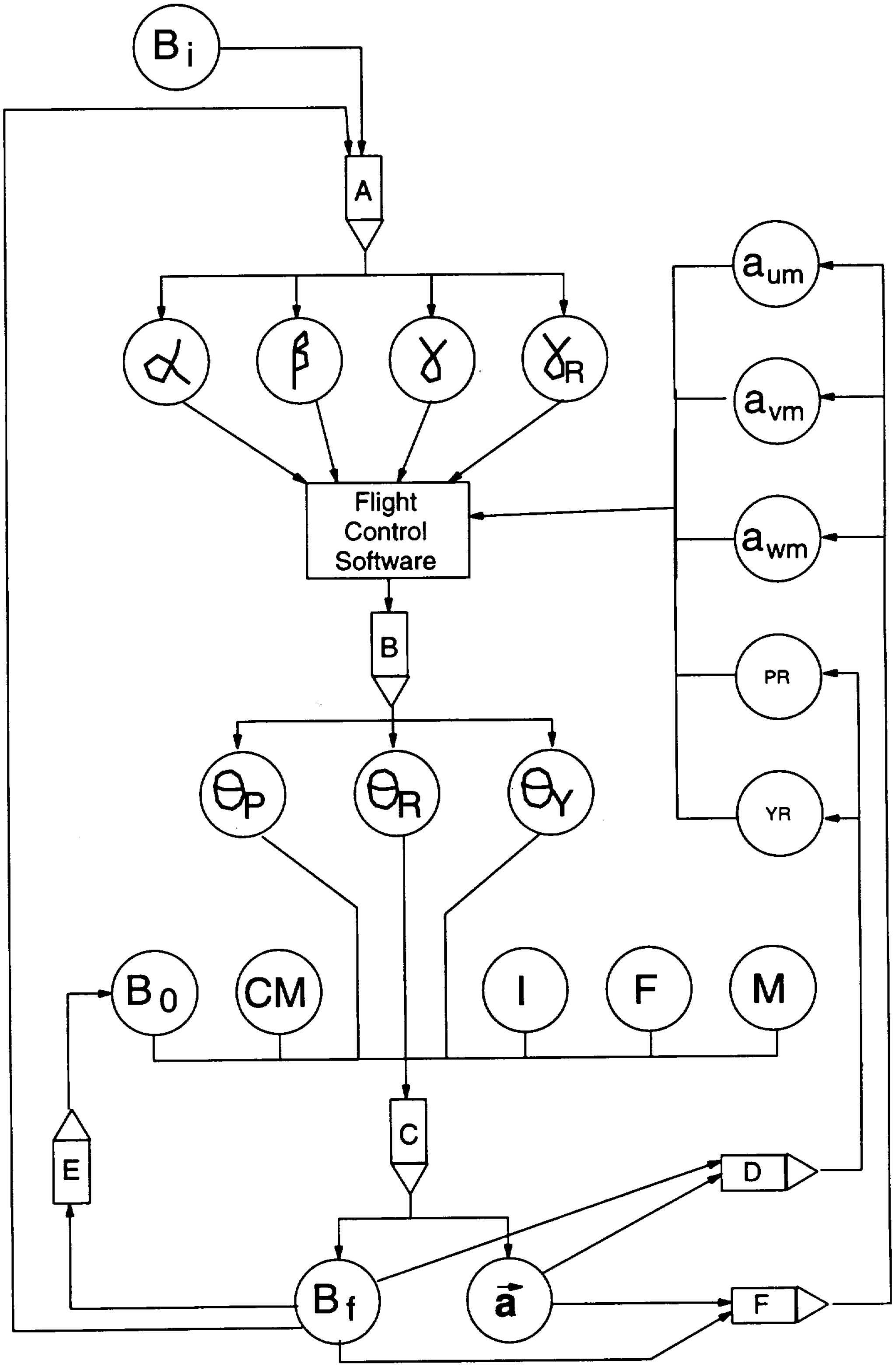


Figure 5.1 : DFM Model of the Titan II Flight Control System

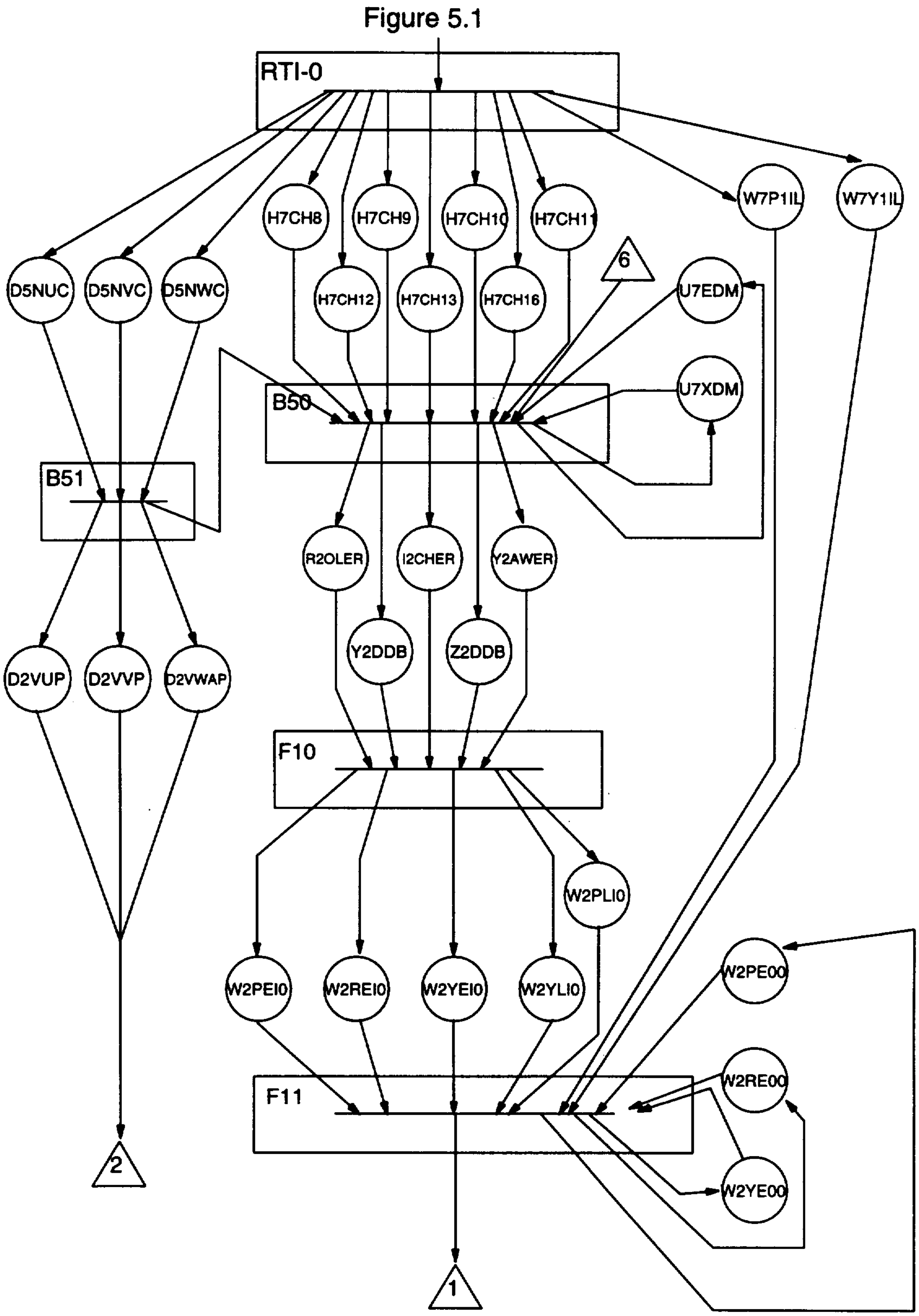


Figure 5.2 : DFM Model of the Flight Control Software (1/3)

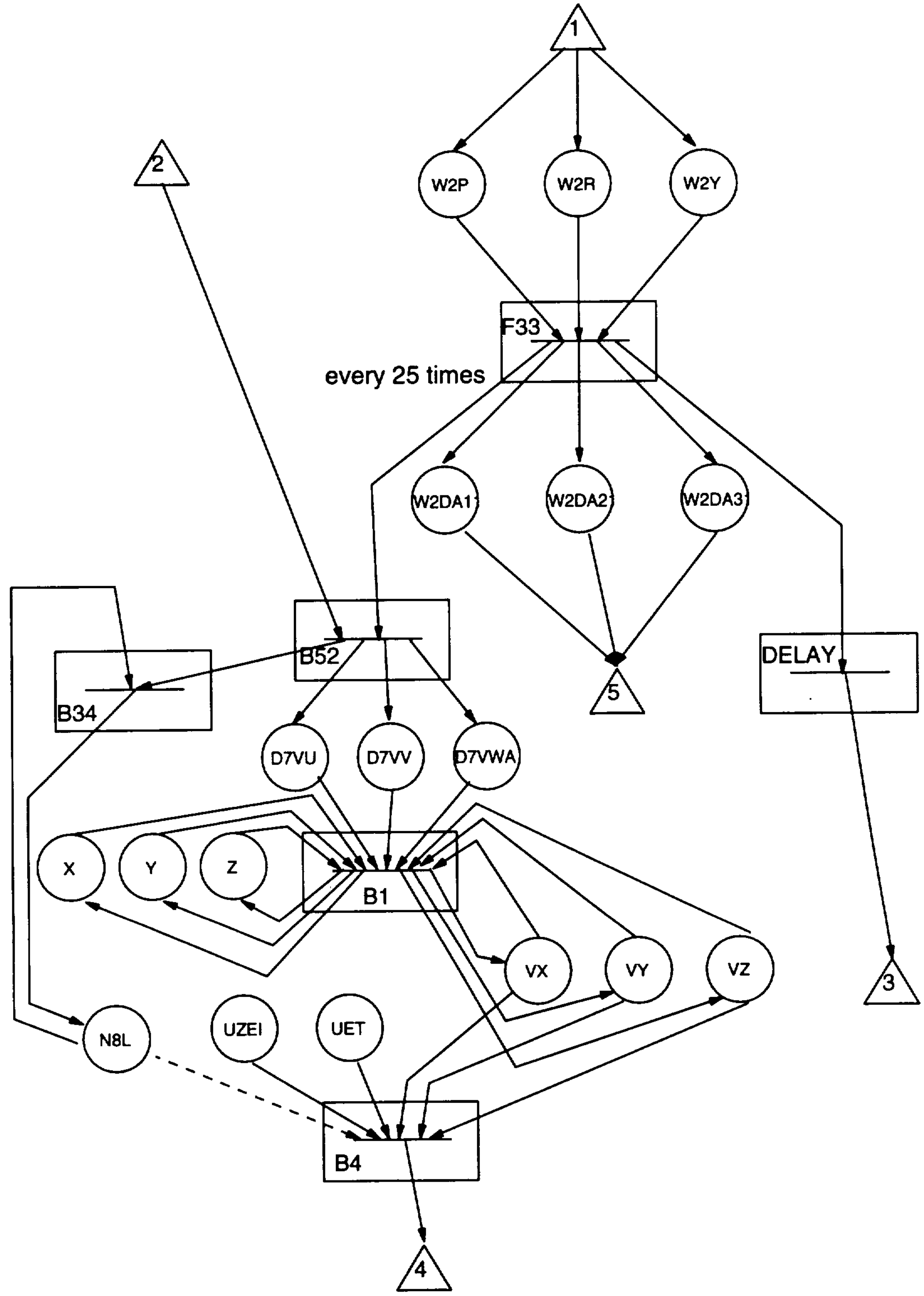


Figure 5.2 : DFM Model of the Flight Control Software (2/3)

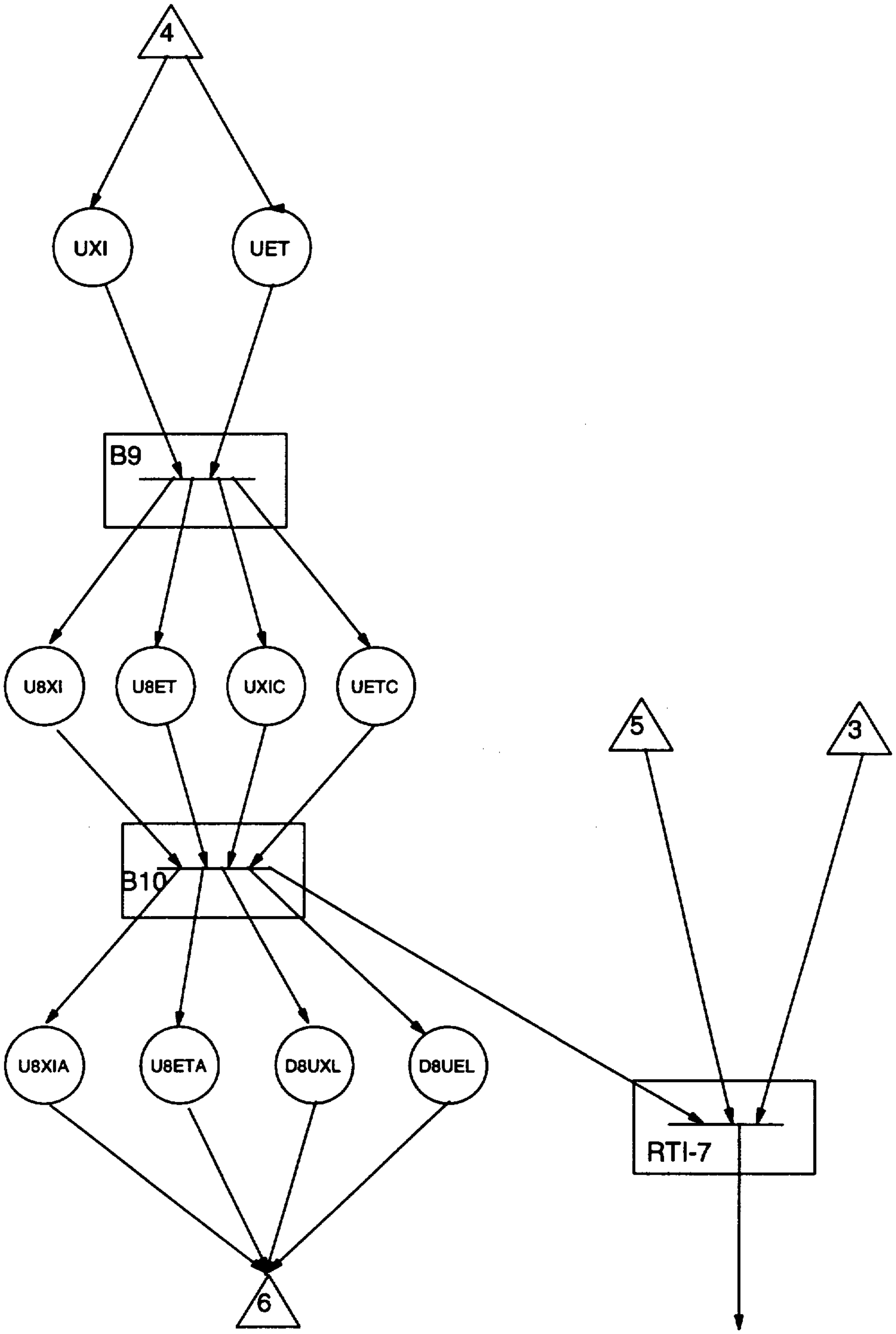


Figure 5.2 : DFM Model of the Flight Control Software (3/3)

updated. The forward loop signals are then converted to D/A outputs in the subroutine FIG 33. The software then waits until the RTI-7 executes to issue the outputs to the actuators. This completes one minor cycle. After executing 25 similar cycles, the time between the completion of FIG 33 and the execution of RTI-7 will be used for major calculations, hence the branching after FIG 33.

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 the 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. X, Y, and Z are each discretized into 5 states, representing a large negative deviation, a moderate deviation, a distance close to 0, a moderate positive deviation, and a large positive deviation. VX, VY, and VZ are also each discretized into 5 states representing a large negative velocity, a moderate velocity, a velocity close to 0, a moderate positive velocity, and a large positive velocity. Similarly, D7VU, D7VV, and D7VWA are each discretized into 5 states representing a large negative acceleration, a moderate negative acceleration, an acceleration close to 0, a moderate positive acceleration, and a large positive acceleration.

The equations implemented by this software module are listed in Table 5.III.

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.	DVSZ	=	RG32*D7VU + RG33*D7VV + RG31*DVW
17.	DVSQ	=	DVSX*DVSX + DVSY+DVSY + DVSZ*DVSZ
18.	X	=	X + VX + 1/2 * (DVSX + DVGX)
19.	Y	=	Y + VY + 1/2 * (DVSY + DVGY)
20.	Z	=	Z + VZ + 1/2 * (DVSZ + DVGZ)

Table 5.III : The Subroutine BLOCK 1 (1/2)

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*CJAG*AG*UZ
30.	RE1	=	DVGX
31.	RE2	=	DVGY
32.	RE3	=	DVGZ
33.	DVGX	=	DVGZE*UX
34.	DVGY	=	DVGZE*UY
35.	DVGZ	=	(DVGJ + DVGJ + DVGZE)*UZ
36.	VX	=	VX + DVSX + 1/2 * (DVGX + RE1)
37.	VY	=	VY + DVSY + 1/2 * (DVGY + RE2)
38.	VZ	=	VZ + DVSZ + 1/2 * (DVGZ + RE3)

Table 5.III : The Subroutine BLOCK 1 (2/2)

Equation 1 calculates the acceleration after compensation for the drift of the IMU. Since we assume no drift, the two accelerations are equal. As D7VU, D7VV, and DVW are not in the same co-ordinate system as X, Y, Z, VX, VY, and VZ, equations 2-16 transform the accelerations into the same co-ordinate system and represent them as DVSX, DVSY, and DVSZ. Equation 17 does nothing more than calculating the square of the magnitude of the acceleration. Equation 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 stores the previous accelerations due to gravity as RE1, RE2, and RE3. Finally, equations 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.

6. A Solution to the Problem of Combinatorial Explosion

The approach to solving the combinatorial explosion problem in the decision table construction is based on the intention to bypass the construction step. It can be observed that the decision tables constructed in Step 1 are only used for providing information for Step 2 (Model Analysis). In the analysis, the causality and the temporal flow of the DFM model is backtracked to identify causes for certain top-events. The decision tables provide intermediate information on the backtracking step, namely in finding the parameter states in-between the top events and their causes. If we are able to find these intermediate causes by some means other than looking up the decision tables, we can avoid the combinatorial explosion 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 Figure 6.1. In the DFM model in Figure 5.1 and Figure 5.2, we find that the condition $(\theta_P, \theta_R, \theta_Y) = (2^\circ, 2.5^\circ, 1.5^\circ)$ is backtracked through the transfer

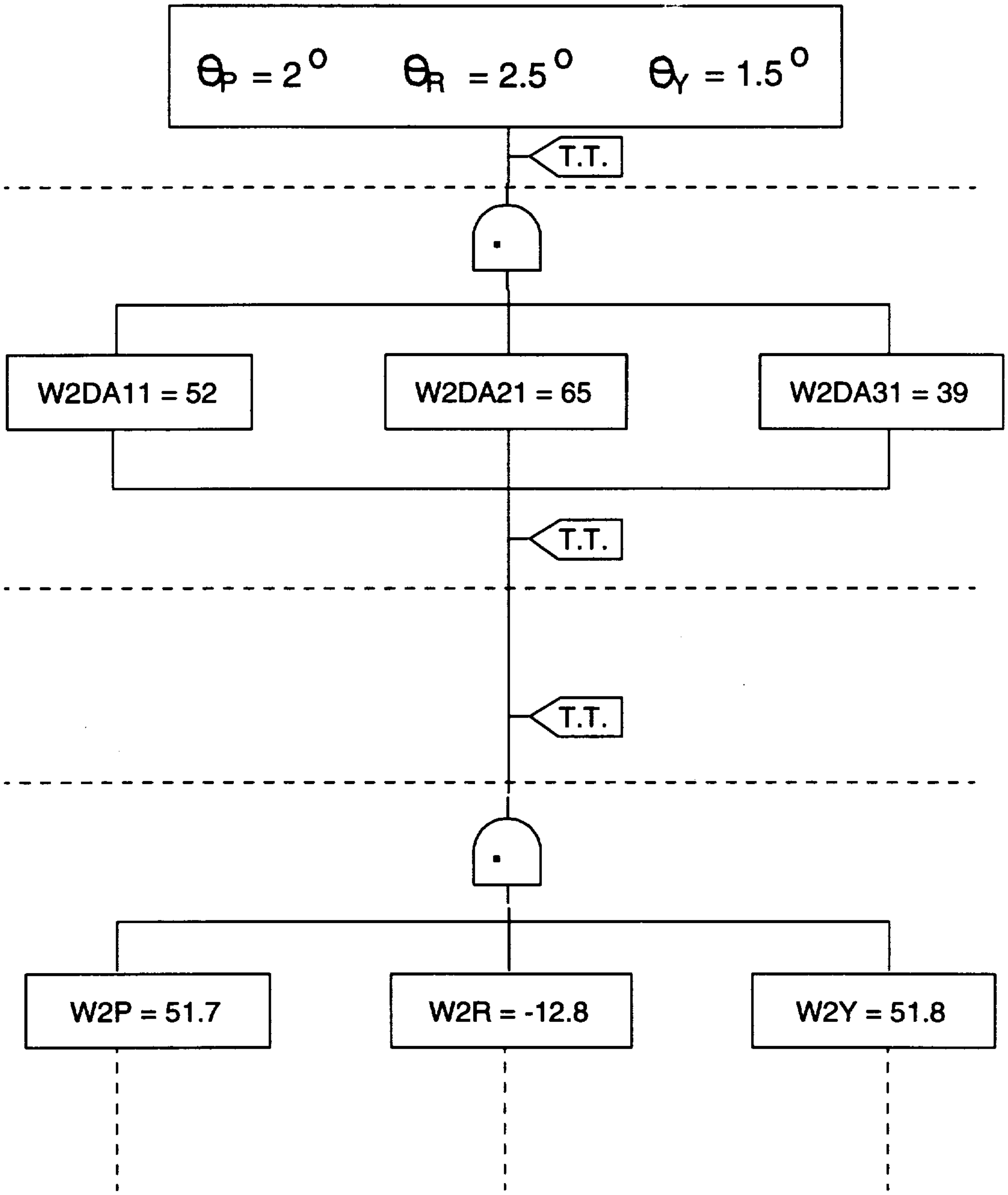


Figure 6.1 : Fault Tree for (θ_P, θ_R, θ_Y) = (2°, 2.5°, 1.5°)

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 = 51.7, W2R = -12.8, and W2Y = 51.8. 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 in 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 [Johnson and Riess, 1982], [Maron, 1987], [Fröberg, 1985] is a well-known approach in numerical analysis for solving a 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.

Given a system of n functions

$$\mathbf{F}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_n(\mathbf{x}))^T$$

where $\mathbf{x} = (x_1, x_2, \ldots, x_n)$,

the Newton-Raphson Method helps us to find the exact solution $\mathbf{a}$ where $\mathbf{F}(\mathbf{a}) = \mathbf{0}$. The

Newton-Raphson Method provides a formula for making successive iteration. This formula is

$\mathbf{x}_{k+1}$ = $\mathbf{x}_k - \mathbf{J}(\mathbf{x}_k)^{-1}\,\mathbf{F}(\mathbf{x}_k)$, where

$\mathbf{x}_{k+1}$ = (k+1)th iteration

$\mathbf{x}_k$ = kth iteration

$\mathbf{J}(\mathbf{x}_k)^{-1}$ = inverse of the Jacobian Matrix calculated using the kth iteration

$\mathbf{J}$ = Jacobian Matrix in which the element in the ith row and jth column is defined as df_i/dx_j

Note that in the iteration, we need to supply an initial guess. This initial guess is crucial to the 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 $x_1, x_2, \ldots, x_n$ and m output variables $y_1, y_2, \ldots, y_m$. The physical equations implemented by this subroutine are:

$$\begin{array}{lcl}
y_1 & = & g_1(x_1, x_2, \ldots, x_n) \\
y_2 & = & g_2(x_1, x_2, \ldots, x_n) \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
y_m & = & g_m(x_1, x_2, \ldots, x_n)
\end{array}$$

In the backtracking analysis, we discover that the outputs $y_1 = b_1, y_2 = b_2, \ldots, y_m = b_m$ 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:

$$\begin{array}{lcl}
f_1(x_1, \ldots, x_n) & = & g_1(x_1, \ldots, x_n) - b_1 \\
f_2(x_1, \ldots, x_n) & = & g_2(x_1, \ldots, x_n) - b_2 \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
f_m(x_1, \ldots, x_n) & = & g_m(x_1, \ldots, x_n) - b_m
\end{array}$$

The objective is then to determine the values $a_1, a_2, \ldots, a_n$ such that

$$\begin{array}{lcl}
f_1(a_1, a_2, \ldots, a_n) & = & 0 \\
f_2(a_1, a_2, \ldots, a_n) & = & 0 \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
f_m(a_1, a_2, \ldots, a_m) & = & 0
\end{array}$$

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 exist 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 $y_1, y_2, ..., y_m$ are not independent. We can choose amongst them the n independent variables. Without loss of generality, let these be $y_1, y_2, ..., y_n$. We can then solve the system

$$\begin{array}{ccc} f_1 & = & 0 \\ . & . & . \\ . & . & . \\ . & . & . \\ f_n & = & 0 \end{array}$$

using Newton-Raphson Method, where

$$f_i(x_1, x_2, ..., x_n) \quad = \quad g_i(x_1, x_2, ..., x_n) - b_i$$

$$i = 1,2,..., n$$

The observation regarding diverging iterations also applies in this situation.

Case 3 m < n

The most interesting case occur 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

$$\begin{array}{ccc} y_1 & = & g_1(x_1, \ldots, x_n) \\ . & . & . \\ . & . & . \\ . & . & . \\ y_m & = & g_m(x_1, \ldots, x_n) \end{array}$$

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

$$\begin{array}{ccc} x_{m+1} & = & k_{m+1} \\ . & . & . \\ . & . & . \\ . & . & . \\ x_n & = & k_n \end{array}$$

We then solve the system

$$f_i \quad = \quad 0 \qquad\qquad i = 1, 2, \ldots, m$$

using the Newton-Raphson Method, where

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

$$i = 1, 2, \ldots, m$$

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

In case a solution does not exist for the assumed arbitrary values

$$\begin{array}{lll} x_{m+1} & = & k_{m+1} \\ . & . & . \\ . & . & . \\ . & . & . \\ x_n & = & k_n \end{array}$$

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. This subroutine is chosen because the problem of combinatorial explosion is encountered in constructing its associated decision table, and the equations implemented in this software module have been discussed in Section 5. 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 previous 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

$$\begin{array}{lll} X & = & 1.001 \text{ x } 10^7 \\ Y & = & 1.001 \text{ x } 10^7 \\ Z & = & 2.002 \text{ x } 10^7 \\ VX & = & -9990 \\ VY & = & 9990 \\ VZ & = & 19980 \end{array}$$

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

D7VU = -100 or 100
D7VV = -100 or 100
D7VWA= -100 or 100

The C program tries the eight possible combinations of D7VU, D7VV, and D7VWA to solve for X, Y, Z, VX, VY, VZ. The code written to implement the approach is shown in Figure 6.2. It uses the initial guesses

$X0=1.001 \times 10^7$
$Y0=1.001 \times 10^7$
$Z0=2.002 \times 10^7$
VX0=-9990
VY0=9990
VZ0=19980

and the results are shown in Figure 6.3. 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.

$X0=-2.1 \times 10^7$
Y0=0
Z0=0
VX0=0
VY0=0
VZ0=0

The results produced are shown in Figure 6.4. 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.

```
/* ******************************************          */
/* header.h                                            */
/* Michael Yau     4-12-1993                           */
/* ******************************************          */

#include <stdio.h>
#include <math.h>

#define MAXITERATION (1000)
#define TOLERANCE (1E-03)
#define CGMN  (-0.14076458E+17)
#define CJAG  (0.25258642E-03)
#define CJG5  (-0.50517284E-04)


#define NUMBER_IN (9)           /* Number of input variables */
#define NUMBER_OUT (6)         /* Number of output variables */
#define NUMBER_SAM (3)         /* Number of input variables to be sampled */
#define SAMPLE_POINT (5)       /* Maximum number of sampling points */
#define h (1)                  /* The step use to calculate Jacobian Matrix */


/* ************************************************  */
/* Define the variables in the main.c file            */
/* ************************************************  */

/* ************************************************  */
/* Define the sampling points in the sample.dat file  */
/* ************************************************  */

void sample( int,int*,int*,double[NUMBER_SAM][SAMPLE_POINT],
double*,double* );

int newton( double*, double* );

void fn( double*, double*, double* );

void jm( double jf[NUMBER_OUT][NUMBER_OUT],
double*, double* );

void check( double* );
```

Figure 6.2 : Software Code for Solving Equations (1/14)

```
/* ******************************************                */
/* main.c                                                    */
/* Solving for the input variables for BLOCK 1               */
/* with a checking algorithm                                 */
/* Michael Yau      4-12-1993                                */
/* ******************************************                */

#include "e:\bcc\newton\header.h"

main()
{
        int i, j, count[NUMBER_SAM];
        int sam_pnt[NUMBER_SAM];
        double x[NUMBER_IN], y[NUMBER_OUT];
        double temp, s[NUMBER_IN][SAMPLE_POINT];
        FILE *sample_in, *data_in;

/*  Keep track of the input and output variables             */
/*                                                           */
/*      x[0] = x0               y[0] = x1                    */
/*      x[1] = y0               y[1] = y1                    */
/*      x[2] = z0               y[2] = z1                    */
/*      x[3] = vx0              y[3] = vx1                   */
/*      x[4] = vy0              y[4] = vy1                   */
/*      x[5] = vz0              y[5] = vz1                   */
/*      x[6] = d7vu             s[0] = d7vu                  */
/*      x[7] = d7vv             s[1] = d7vv                  */
/*      x[8] = d7vwz            s[2] = d7vwz                 */
/*                                                           */
/* *******************************************               */

/* Get the values of the output variables from input.dat */

        data_in = fopen("input.dat","r");

/* Get the values in the order x1,y1,z1,vx1,vy1,vz1 */

        while ( fscanf(data_in,"%lf",&y[0]) == 1 ){
```

Figure 6.2 : Software Code for Solving Equations (2/14)

```
                for ( i = 1; i < NUMBER_OUT; i++ ){
                        fscanf(data_in,"%lf",&temp);
                        y[i] = temp;
                }

                printf("\n**************************************\n\n");
                printf("For the output variables:\n\n");
                for ( i = 0; i < NUMBER_OUT; i++ ){
                        printf("y[%2d] = %10.3e\n",i,y[i]);
                }
        /* Read in the sampling points */

                sample_in = fopen("sampnt.dat","r");
                for ( i = 0; i < NUMBER_SAM; i++ ){
                        fscanf(sample_in,"%d",&sam_pnt[i]);
                        for ( j = 0; j < sam_pnt[i]; j++ ){
                                fscanf(sample_in,"%lf",&temp);
                                s[i][j] = temp;
                        }
                }
                fclose(sample_in);

        /* Start sampling */

                sample( 0, count, sam_pnt, s, x, y );
        }
        fclose(data_in);
        return NULL;
}
```

Figure 6.2 : Software Code for Solving Equations (3/14)

```
/* ******************************************          */
/* sample.c                                             */
/* Michael Yau     4-12-1993                            */
/* ******************************************          */

#include "e:\bcc\newton\header.h"

void sample(int level,int count[NUMBER_SAM],int sam_pnt[NUMBER_SAM],
            double s[NUMBER_SAM][SAMPLE_POINT],
            double x[NUMBER_IN], double y[NUMBER_OUT])
{
        int i, j, newton_return;
        for ( count[level]=0; count[level]<sam_pnt[level]; count[level]++ ){
                if ( level < NUMBER_SAM-1 )
                        sample( level+1, count, sam_pnt, s, x, y );
                else
                {
                        printf("\n****************************\n");
                        printf("\n      For the sample points:\n\n");
                        for ( i = NUMBER_OUT; i < NUMBER_IN; i++ ){
                                j = count[i-NUMBER_OUT];
                                x[i] = s[i-NUMBER_OUT][j];
                                printf("     x[%2d] = %10.3e\n",i,x[i]);
                        }
                        newton_return = newton( x, y );
                        if ( newton_return == 0 )
                        {
                                printf("\n\n           No solution for this set of sample
                                points\n");
                        }
                        else
                        {
                                printf("\n\n           The solution is:\n\n");
                                for( i = 0; i < NUMBER_IN; i++ ){
                                        printf("           x[%2d] = %10.3e\n",i,x[i]);
                                }
                                check( x );
                        }
                }
        }
}
```

Figure 6.2 : Software Code for Solving Equations (4/14)

```
/* *************************          */
/* newton.c                           */
/* Michael Yau    4-12-1993           */
/* *************************          */

#include "e:\bcc\newton\header.h"

/* *** Subroutine to implement Newton's Method *** */

int newton( double*x, double*y )
{
        double f[NUMBER_OUT], jf[NUMBER_OUT][NUMBER_OUT];
        double dx[NUMBER_OUT], m[NUMBER_OUT][NUMBER_OUT];
        double previous_error, error, ftemp, jftemp, jfmax;
        int count, i, j, k, l, imax;

        double find_error( double* );

        /* ******************************     */
        /* initialize the solution vector     */
        /* ******************************     */

        x[0] = -2.1e+07;
        x[1] = 0;
        x[2] = 0;
        x[3] = 0;
        x[4] = 0;
        x[5] = 0;

        printf("\n%4s %9s %9s %9s %9s %9s %9s %9s\n",
                "N","x0","y0","z0","vx0","vy0","vz0","Error");

        /* Start iteration */

        for ( count = 0; count < MAXITERATION; count++ ){
```

Figure 6.2 : Software Code for Solving Equations (5/14)

```
/* Print out the guess */

printf("\n%4d",count);
for ( i = 0; i < NUMBER_OUT; i++ ){
        printf(" %9.2e",x[i]);
}

/* Evaluate f and reverse the sign */

fn( f, x, y );

/* Find the error */

error = find_error( f );
printf(" %9.2e",error);
if ( error < TOLERANCE )
        return 1;
if ( count > 0 )
        if ( error > previous_error )
                return 0;
previous_error = error;

/* Calculate the Jacobian Matrix */

jm( jf, x, y );

/* Calculate the inverse of the Jacobian Matrix */

for( k = 0; k < NUMBER_OUT; k++){

        /* Row Interchange */

        imax = k;
        jfmax = fabs(jf[k][k]);
        for( l = k+1; l < NUMBER_OUT; l++){
                if ( fabs(jf[l][k]) > jfmax )
                {
                        jfmax = fabs(jf[l][k]);
                        imax = l;
                }
        }
```

Figure 6.2 : Software Code for Solving Equations (6/14)

```
        if ( imax != k )
        {
                for( l = k; l < NUMBER_OUT; l++){
                        jftemp = jf[k][l];
                        jf[k][l] = jf[imax][l];
                        jf[imax][l] = jftemp;
                }
                ftemp = f[k];
                f[k] = f[imax];
                f[imax] = ftemp;
        }

        /* Forward Elimination */

        for( i = k+1; i < NUMBER_OUT; i++){
                m[i][k] = jf[i][k] / jf[k][k];
                f[i] -= m[i][k] * f[k];
                for( j = k+1; j < NUMBER_OUT; j++){
                        jf[i][j] -= m[i][k] * jf[k][j];
                }
        }
}

/* Back Substitution */

dx[NUMBER_OUT-1] = f[NUMBER_OUT-1] /
jf[NUMBER_OUT-1][NUMBER_OUT-1];
for( i = NUMBER_OUT-2; i >= 0; i--){
        dx[i] = 0.0;
        for( j = i+1; j < NUMBER_OUT; j++){
                dx[i] += jf[i][j] * dx[j];
        }
        dx[i] = ( f[i] - dx[i] ) / jf[i][i];
}
```

Figure 6.2 : Software Code for Solving Equations (7/14)

```
        /* Calculate the next guesses */

        for ( i = 0; i < NUMBER_OUT; i++ ){
            x[i] += dx[i];
        }
    }
    return 1;
}

/* *** End of Subroutine newton *** */


/* *** Subroutine to find the error *** */

double find_error( double* f )
{
    double error=0;
    int i;

    for ( i = 0; i < NUMBER_OUT; i++ ){
        error += f[i]*f[i];
    }
    return sqrt(error);
}

/* *** End of Subroutine find_error *** */
```

Figure 6.2 : Software Code for Solving Equations (8/14)

```
/* ********************************************        */
/* function.c                                          */
/* This file contains the function definitions for     */
/* the subroutine                                      */
/* Michael Yau 4-12-1993                               */
/* ********************************************        */

#include "e:\bcc\newton\header.h"

/* *** Subroutine to evaluate the function *** */

void fn( double* f, double* x, double* y )
{
        int i;
        double CG11, CG12, CG13, CG21, CG22, CG23,  CG31, CG32, CG33;
        double D7VU, D7VV, D7VWA, DVW, DVSQ;
        double X, Y, Z, VX, VY, VZ, DVSX, DVSY, DVSZ;
        double DVGX, DVGY, DVGZ, RE1, RE2, RE3;
        double FI11, FI12, FI13, FI21, FI22, FI23,  FI31, FI32, FI33;
        double RG11, RG12, RG13, RG21, RG22, RG23, RG31, RG32, RG33;
        double p[3];
        void dvg( double*, double*, double*, double* );

        X               = x[0];
        Y               = x[1];
        Z               = x[2];
        VX              = x[3];
        VY              = x[4];
        VZ              = x[5];
        D7VU            = x[6];
        D7VV            = x[7];
        D7VWA           = x[8];
        FI11 = 1;
        FI12 = 0;
        FI13 = 0;
        FI21 = 0;
        FI22 = 1;
        FI23 = 0;
```

Figure 6.2 : Software Code for Solving Equations (9/14)

```
CG11 =  0.14379912;
CG12 = -0.86644896;
CG13 =  0.47810879;
CG21 =  0.09714823;
CG22 =  0.49315612;
CG23 =  0.86449942;
CG31 = -0.98482691;
CG32 = -0.07786683;
CG33 =  0.15508940;
DVW = D7VWA;
FI31 = FI12*FI23 - FI22*FI13;
FI32 = FI13*FI21 - FI23*FI11;
FI33 = FI11*FI22 - FI21*FI12;
RG11 = CG21*FI13 + CG31*FI23 + CG11*FI33;
RG21 = CG22*FI13 + CG32*FI23 + CG12*FI33;
RG31 = CG23*FI13 + CG33*FI23 + CG13*FI33;
RG12 = CG21*FI11 + CG31*FI21 + CG11*FI31;
RG22 = CG22*FI11 + CG32*FI21 + CG12*FI31;
RG32 = CG23*FI11 + CG33*FI21 + CG13*FI31;
RG13 = RG21*RG32 - RG22*RG31;
RG23 = RG31*RG12 - RG32*RG11;
RG33 = RG11*RG22 - RG12*RG21;
DVSX = RG12*D7VU + RG13*D7VV + RG11*DVW;
DVSY = RG22*D7VU + RG23*D7VV + RG21*DVW;
DVSZ = RG32*D7VU + RG33*D7VV + RG31*DVW;
DVSQ = DVSX*DVSX + DVSY*DVSY + DVSZ*DVSZ;
p[0] = X;
p[1] = Y;
p[2] = Z;
dvg( &DVGX, &DVGY, &DVGZ, p );
X = X + VX + 0.5*( DVSX + DVGX );
Y = Y + VY + 0.5*( DVSY + DVGY );
Z = Z + VZ + 0.5*( DVSZ + DVGZ );
RE1 = DVGX;
RE2 = DVGY;
RE3 = DVGZ;
p[0] = X;
p[1] = Y;
p[2] = Z;
dvg( &DVGX, &DVGY, &DVGZ, p );
```

Figure 6.2 : Software Code for Solving Equations (10/14)

```
        VX = VX + DVSX + 0.5*( DVGX + RE1 );
        VY = VY + DVSY + 0.5*( DVGY + RE2 );
        VZ = VZ + DVSZ + 0.5*( DVGZ + RE3 );
        f[0] = X - y[0];
        f[1] = Y - y[1];
        f[2] = Z - y[2];
        f[3] = VX - y[3];
        f[4] = VY - y[4];
        f[5] = VZ - y[5];
        for ( i = 0; i < NUMBER_OUT; i++ ){
                f[i] = -f[i];
        }
}

/* *** End of Subroutine fn *** */



/* *** Subroutine to find dvgx, dvgy, and dvgz *** */

void dvg( double *dvgx, double *dvgy, double *dvgz,
double p[3] )
{
        double r, ux, uy, uz, ag, dvgj, dvgze;

        r = sqrt( (p[0]*p[0] + p[1]*p[1] + p[2]*p[2]) );
        ux = p[0]/r;
        uy = p[1]/r;
        uz = p[2]/r;
        ag = CGMN / (r*r);
        dvgj = ag * CJG5 * ag;
        dvgze = ag + dvgj + ag*uz*CJAG*uz*ag;
        *dvgx = dvgze * ux;
        *dvgy = dvgze * uy;
        *dvgz = ( 2*dvgj + dvgze ) * uz;
}

/* *** End of Subroutine dvg *** */
```

Figure 6.2 : Software Code for Solving Equations (11/14)

```
/* *** Subroutine to find the Jacobian Matrix *** */

void jm( double jf[NUMBER_OUT][NUMBER_OUT], double* x, double* y )
{
        double f_plus[NUMBER_OUT], f_minus[NUMBER_OUT];
        double x_plus[NUMBER_IN], x_minus[NUMBER_IN];
        int i, j;
        for ( i = 0; i < NUMBER_OUT; i++ ){
                for ( j = 0; j < NUMBER_IN; j++ ){
                        if ( j == i )
                        {
                                x_plus[j]  = x[j] + h;
                                x_minus[j] = x[j] - h;
                        }
                        else
                        {
                                x_plus[j]  = x[j];
                                x_minus[j] = x[j];
                        }
                }
                fn( f_plus , x_plus , y );
                fn( f_minus, x_minus, y );
                for ( j = 0; j < NUMBER_OUT; j++ ){
                        jf[j][i] = ( f_minus[j]-f_plus[j] ) / (2*h);
                }
        }
}

/* *** End of Subroutine jm *** */
```

Figure 6.2 : Software Code for Solving Equations (12/14)

```
/* ******************************    */
/* check.c                           */
/* Michael Yau    4-14-1993          */
/* ******************************    */

#include "e:\bcc\newton\header.h"

/* *** Subroutine to check the solution *** */

void check( double*x )
{
        double z[NUMBER_OUT];
        double dvgx0, dvgy0, dvgz0, dvgx1, dvgy1, dvgz1;
        double CG11, CG12, CG13, CG21, CG22, CG23;
        double RG11, RG12, RG13, RG21, RG22, RG23;
        double RG31, RG32, RG33;
        double DVSX, DVSY, DVSZ;
        void dvg( double*, double*, double*, double* );

        CG11 =  0.14379912;
        CG12 = -0.86644896;
        CG13 =  0.47810879;
        CG21 =  0.09714823;
        CG22 =  0.49315612;
        CG23 =  0.86449942;
        RG11 = CG11;
        RG21 = CG12;
        RG31 = CG13;
        RG12 = CG21;
        RG22 = CG22;
        RG32 = CG23;
        RG13 = RG21*RG32 - RG22*RG31;
        RG23 = RG31*RG12 - RG32*RG11;
        RG33 = RG11*RG22 - RG12*RG21;
        DVSX = RG12*x[6] + RG13*x[7] + RG11*x[8];
        DVSY = RG22*x[6] + RG23*x[7] + RG21*x[8];
        DVSZ = RG32*x[6] + RG33*x[7] + RG31*x[8];
        dvg( &dvgx0, &dvgy0, &dvgz0, x );
```

Figure 6.2 : Software Code for Solving Equations (13/14)

```
    z[0] = x[0] + x[3] + 0.5*( DVSX + dvgx0 );
    z[1] = x[1] + x[4] +    5*( DVSY + dvgy0 );
    z[2] = x[2] + x[5] + 0.5*( DVSZ + dvgz0 );

    dvg( &dvgx1, &dvgy1, &dvgz1, z );

    z[3] = x[3] + DVSX + 0.5*( dvgx0 + dvgx1 );
    z[4] = x[4] + DVSY + 0.5*( dvgy0 + dvgy1 );
    z[5] = x[5] + DVSZ + 0.5*( dvgz0 + dvgz1 );

    /* Print out the result of checking */

    printf("\n          Result of Checking:\n");
    printf("\n          x1 = %10.3e  y1 = %10.3e  z1 =  %10.3e",z[0],z[1],z[2]);
    printf("\n          vx1 = %10.3e  vy1 = %10.3e  vz1 =
    %10.3e\n",z[3],z[4],z[5]);
}

/* *** End of Subroutine check *** */
```

Figure 6.2 : Software Code for Solving Equations (14/14)

```
*************************************
For the output variables:
y[ 0] =  1.001e+07
y[ 1] =  1.001e+07
y[ 2] =  2.002e+07
y[ 3] = -9.990e+03
y[ 4] =  9.990e+03
y[ 5] =  1.998e+04
    ****************************
    For the sample points:
    x[ 6] = -1.000e+02
    x[ 7] = -1.000e+02
    x[ 8] = -1.000e+02

Iteration

   0         X  =  1.00e+07Y  =  1.00e+07Z  =  2.00e+07
             VX = -9.90e+03VY =  9.90e+03VZ =  2.00e+04
             Error =  2.43e+04

   1         X  =  1.00e+07Y  =  1.00e+07Z  =  2.00e+07
             VX = -9.88e+03VY =  1.01e+04VZ =  2.01e+04
             Error =  2.61e-05

        The solution is:

        x[ 0] =  1.002e+07
        x[ 1] =  1.000e+07
        x[ 2] =  2.000e+07
        x[ 3] = -9.880e+03
        x[ 4] =  1.010e+04
        x[ 5] =  2.010e+04
        x[ 6] = -1.000e+02
        x[ 7] = -1.000e+02
        x[ 8] = -1.000e+02

        Result of Checking:

        x1  = 1.001e+07  y1  = 1.001e+07  z1 = 2.002e+07
        vx1 = -9.990e+03 vy1 = 9.990e+03 vz1 = 1.998e+04
```

Figure 6.3 : Result of Iteration (1/8)

```
*****************************

    For the sample points:

    x[ 6] = -1.000e+02
    x[ 7] = -1.000e+02
    x[ 8] =  1.000e+02

Iteration

   0          X  =  1.00e+07      Y  =  1.00e+07      Z  =  2.00e+07
              VX = -9.90e+03      VY =  9.90e+03      VZ =  2.00e+04
              Error =  2.44e+04

   1          X  =  1.00e+07      Y  =  1.00e+07      Z  =  2.00e+07
              VX = -9.88e+03      VY =  1.01e+04      VZ =  1.99e+04
              Error =  2.59e-05


        The solution is:

        x[ 0] =  1.002e+07
        x[ 1] =  1.000e+07
        x[ 2] =  2.000e+07
        x[ 3] = -9.880e+03
        x[ 4] =  1.010e+04
        x[ 5] =  1.990e+04
        x[ 6] = -1.000e+02
        x[ 7] = -1.000e+02
        x[ 8] =  1.000e+02

        Result of Checking:

        x1  = 1.001e+07  y1  = 1.001e+07  z1  = 2.002e+07
        vx1 = -9.990e+03 vy1 = 9.990e+03  vz1 = 1.998e+04
```

Figure 6.3 : Result of Iteration (2/8)

```
    ****************************

    For the sample points:

    x[ 6] = -1.000e+02
    x[ 7] =  1.000e+02
    x[ 8] = -1.000e+02

Iteration

    0          X  =  1.00e+07      Y  =  1.00e+07      Z  =  2.00e+07
               VX = -9.90e+03      VY =  9.90e+03      VZ =  2.00e+04
               Error =  2.44e+04

    1          X  =  1.00e+07      Y  =  1.00e+07      Z  =  2.00e+07
               VX = -9.88e+03      VY =  9.90e+03      VZ =  2.01e+04
               Error =  2.60e-05


        The solution is:

        x[ 0] =  1.002e+07
        x[ 1] =  1.000e+07
        x[ 2] =  2.000e+07
        x[ 3] = -9.880e+03
        x[ 4] =  9.900e+03
        x[ 5] =  2.010e+04
        x[ 6] = -1.000e+02
        x[ 7] =  1.000e+02
        x[ 8] = -1.000e+02

        Result of Checking:

        x1  = 1.001e+07  y1  = 1.001e+07  z1  = 2.002e+07
        vx1 = -9.990e+03 vy1 = 9.990e+03  vz1 = 1.998e+04
```

Figure 6.3 : Result of Iteration (3/8)

```
    *****************************

    For the sample points:

    x[ 6] = -1.000e+02
    x[ 7] =  1.000e+02
    x[ 8] =  1.000e+02

Iteration

   0         X  =  1.00e+07      Y  =  1.00e+07      Z  =  2.00e+07
             VX = -9.90e+03      VY =  9.90e+03      VZ =  2.00e+04
             Error =  2.45e+04

   1         X  =  1.00e+07      Y  =  1.00e+07      Z  =  2.00e+07
             VX = -9.88e+03      VY =  9.90e+03      VZ =  1.99e+04
             Error =  2.57e-05


      The solution is:

      x[ 0] =  1.002e+07
      x[ 1] =  1.000e+07
      x[ 2] =  2.000e+07
      x[ 3] = -9.880e+03
      x[ 4] =  9.900e+03
      x[ 5] =  1.990e+04
      x[ 6] = -1.000e+02
      x[ 7] =  1.000e+02
      x[ 8] =  1.000e+02

      Result of Checking:

      x1  = 1.001e+07  y1  = 1.001e+07 z1  = 2.002e+07
      vx1 = -9.990e+03 vy1 = 9.990e+03 vz1 = 1.998e+04
```

Figure 6.3 : Result of Iteration (4/8)

```
    *****************************

    For the sample points:

    x[ 6] =  1.000e+02
    x[ 7] = -1.000e+02
    x[ 8] = -1.000e+02

Iteration

   0        X  =  1.00e+07      Y  =  1.00e+07      Z  =  2.00e+07
            VX = -9.90e+03      VY =  9.90e+03      VZ =  2.00e+04
            Error =  2.43e+04

   1        X  =  1.00e+07      Y  =  1.00e+07      Z  =  2.00e+07
            VX = -1.01e+04      VY =  1.01e+04      VZ =  2.01e+04
            Error =  2.61e-05


        The solution is:

        x[ 0] =  1.002e+07
        x[ 1] =  1.000e+07
        x[ 2] =  2.000e+07
        x[ 3] = -1.008e+04
        x[ 4] =  1.010e+04
        x[ 5] =  2.010e+04
        x[ 6] =  1.000e+02
        x[ 7] = -1.000e+02
        x[ 8] = -1.000e+02

        Result of Checking:

        x1  = 1.001e+07  y1  = 1.001e+07  z1  = 2.002e+07
        vx1 = -9.990e+03 vy1 = 9.990e+03  vz1 = 1.998e+04
```

Figure 6.3 : Result of Iteration (5/8)

```
    ****************************

    For the sample points:

    x[ 6] =  1.000e+02
    x[ 7] = -1.000e+02
    x[ 8] =  1.000e+02

Iteration

    0         X  =  1.00e+07       Y  =  1.00e+07       Z  =  2.00e+07
              VX = -9.90e+03       VY =  9.90e+03       VZ =  2.00e+04
              Error =  2.44e+04

    1         X  =  1.00e+07       Y  =  1.00e+07       Z  =  2.00e+07
              VX = -1.01e+04       VY =  1.01e+04       VZ =  1.99e+04
              Error =  2.58e-05


        The solution is:

        x[ 0] =  1.002e+07
        x[ 1] =  1.000e+07
        x[ 2] =  2.000e+07
        x[ 3] = -1.008e+04
        x[ 4] =  1.010e+04
        x[ 5] =  1.990e+04
        x[ 6] =  1.000e+02
        x[ 7] = -1.000e+02
        x[ 8] =  1.000e+02

        Result of Checking:

        x1  = 1.001e+07  y1  = 1.001e+07  z1  = 2.002e+07
        vx1 = -9.990e+03 vy1 = 9.990e+03  vz1 = 1.998e+04
```

Figure 6.3 : Result of Iteration (6/8)

```
    ****************************

    For the sample points:

    x[ 6] =  1.000e+02
    x[ 7] =  1.000e+02
    x[ 8] = -1.000e+02

Iteration

   0          X  =  1.00e+07      Y  =  1.00e+07      Z  =  2.00e+07
              VX = -9.90e+03      VY =  9.90e+03      VZ =  2.00e+04
              Error =  2.43e+04

   1          X  =  1.00e+07      Y  =  1.00e+07      Z  =  2.00e+07
              VX = -1.01e+04      VY =  9.90e+03      VZ =  2.01e+04
              Error =  2.59e-05


        The solution is:

        x[ 0] =  1.002e+07
        x[ 1] =  1.000e+07
        x[ 2] =  2.000e+07
        x[ 3] = -1.008e+04
        x[ 4] =  9.900e+03
        x[ 5] =  2.010e+04
        x[ 6] =  1.000e+02
        x[ 7] =  1.000e+02
        x[ 8] = -1.000e+02

        Result of Checking:

        x1  = 1.001e+07  y1  = 1.001e+07  z1  = 2.002e+07
        vx1 = -9.990e+03 vy1 = 9.990e+03  vz1 = 1.998e+04
```

Figure 6.3 : Result of Iteration (7/8)

```
    ***************************

    For the sample points:

    x[ 6] =  1.000e+02
    x[ 7] =  1.000e+02
    x[ 8] =  1.000e+02

Iteration

  0          X  =  1.00e+07       Y  =  1.00e+07       Z  =  2.00e+07
             VX = -9.90e+03       VY =  9.90e+03       VZ =  2.00e+04
             Error =  2.44e+04

  1          X  =  1.00e+07       Y  =  1.00e+07       Z  =  2.00e+07
             VX = -1.01e+04       VY =  9.90e+03       VZ =  1.99e+04
             Error =  2.57e-05


        The solution is:

        x[ 0] =  1.002e+07
        x[ 1] =  1.000e+07
        x[ 2] =  2.000e+07
        x[ 3] = -1.008e+04
        x[ 4] =  9.900e+03
        x[ 5] =  1.990e+04
        x[ 6] =  1.000e+02
        x[ 7] =  1.000e+02
        x[ 8] =  1.000e+02

        Result of Checking:

        x1  = 1.001e+07  y1  = 1.001e+07  z1  = 2.002e+07
        vx1 = -9.990e+03 vy1 = 9.990e+03  vz1 = 1.998e+04
```

Figure 6.3 : Result of Iteration (8/8)

```
*************************************
For the output variables:
y[ 0] =  1.001e+07
y[ 1] =  1.001e+07
y[ 2] =  2.002e+07
y[ 3] = -9.990e+03
y[ 4] =  9.990e+03
y[ 5] =  1.998e+04
    ****************************
    For the sample points:
    x[ 6] = -1.000e+02
    x[ 7] = -1.000e+02
    x[ 8] = -1.000e+02

Iteration

   0          X  = -2.10e+07      Y  =  0.00e+00      Z  =  0.00e+00
              VX =  0.00e+00      VY =  0.00e+00      VZ =  0.00e+00
              Error =  3.82e+07
   1          X  =  1.00e+07      Y  =  1.00e+07      Z  =  2.00e+07
              VX = -9.95e+03      VY =  1.01e+04      VZ =  2.01e+04
              Error =  9.67e+01
   2          X  =  1.00e+07      Y  =  1.00e+07      Z  =  2.00e+07
              VX = -9.88e+03      VY =  1.01e+04      VZ =  2.01e+04
              Error =  9.23e-10

        The solution is:
        x[ 0] =  1.002e+07
        x[ 1] =  1.000e+07
        x[ 2] =  2.000e+07
        x[ 3] = -9.880e+03
        x[ 4] =  1.010e+04
        x[ 5] =  2.010e+04
        x[ 6] = -1.000e+02
        x[ 7] = -1.000e+02
        x[ 8] = -1.000e+02

        Result of Checking:
        x1  = 1.001e+07  y1  = 1.001e+07  z1  = 2.002e+07
        vx1 = -9.990e+03 vy1 = 9.990e+03  vz1 = 1.998e+04
```

Figure 6.4 : Result for a Different Initial Guess (1/8)

```
     ****************************

     For the sample points:

     x[ 6] = -1.000e+02
     x[ 7] = -1.000e+02
     x[ 8] =  1.000e+02

Iteration

  0          X  = -2.10e+07      Y  =  0.00e+00      Z  =  0.00e+00
             VX =  0.00e+00      VY =  0.00e+00      VZ =  0.00e+00
             Error =  3.82e+07

  1          X  =  1.00e+07      Y  =  1.00e+07      Z  =  2.00e+07
             VX = -9.95e+03      VY =  1.01e+04      VZ =  1.99e+04
             Error =  9.67e+01

  2          X  =  1.00e+07      Y  =  1.00e+07      Z  =  2.00e+07
             VX = -9.88e+03      VY =  1.01e+04      VZ =  1.99e+04
             Error =  9.23e-10


        The solution is:

        x[ 0] =  1.002e+07
        x[ 1] =  1.000e+07
        x[ 2] =  2.000e+07
        x[ 3] = -9.880e+03
        x[ 4] =  1.010e+04
        x[ 5] =  1.990e+04
        x[ 6] = -1.000e+02
        x[ 7] = -1.000e+02
        x[ 8] =  1.000e+02

        Result of Checking:

        x1  = 1.001e+07  y1  = 1.001e+07  z1  = 2.002e+07
        vx1 = -9.990e+03 vy1 = 9.990e+03  vz1 = 1.998e+04
```

Figure 6.4 : Result for a Different Initial Guess (2/8)

```
    ****************************

    For the sample points:

    x[ 6] = -1.000e+02
    x[ 7] =  1.000e+02
    x[ 8] = -1.000e+02

Iteration

   0          X  = -2.10e+07     Y  =  0.00e+00     Z  =  0.00e+00
              VX =  0.00e+00     VY =  0.00e+00     VZ =  0.00e+00
              Error =  3.82e+07

   1          X  =  1.00e+07     Y  =  1.00e+07     Z  =  2.00e+07
              VX = -9.95e+03     VY =  9.90e+03     VZ =  2.01e+04
              Error =  9.67e+01

   2          X  =  1.00e+07     Y  =  1.00e+07     Z  =  2.00e+07
              VX = -9.88e+03     VY =  9.90e+03     VZ =  2.01e+04
              Error =  9.19e-10


        The solution is:

        x[ 0] =  1.002e+07
        x[ 1] =  1.000e+07
        x[ 2] =  2.000e+07
        x[ 3] = -9.880e+03
        x[ 4] =  9.900e+03
        x[ 5] =  2.010e+04
        x[ 6] = -1.000e+02
        x[ 7] =  1.000e+02
        x[ 8] = -1.000e+02

        Result of Checking:

        x1  = 1.001e+07  y1  = 1.001e+07  z1  = 2.002e+07
        vx1 = -9.990e+03 vy1 = 9.990e+03  vz1 = 1.998e+04
```

Figure 6.4 : Result for a Different Initial Guess (3/8)

```
     ****************************

     For the sample points:

     x[ 6] = -1.000e+02
     x[ 7] =  1.000e+02
     x[ 8] =  1.000e+02

Iteration

     0          X  = -2.10e+07      Y  =  0.00e+00      Z  =  0.00e+00
                VX =  0.00e+00      VY =  0.00e+00      VZ =  0.00e+00
                Error =  3.82e+07

     1          X  =  1.00e+07      Y  =  1.00e+07      Z  =  2.00e+07
                VX = -9.95e+03      VY =  9.90e+03      VZ =  1.99e+04
                Error =  9.67e+01

     2          X  =  1.00e+07      Y  =  1.00e+07      Z  =  2.00e+07
                VX = -9.88e+03      VY =  9.90e+03      VZ =  1.99e+04
                Error =  9.21e-10

        The solution is:

        x[ 0] =  1.002e+07
        x[ 1] =  1.000e+07
        x[ 2] =  2.000e+07
        x[ 3] = -9.880e+03
        x[ 4] =  9.900e+03
        x[ 5] =  1.990e+04
        x[ 6] = -1.000e+02
        x[ 7] =  1.000e+02
        x[ 8] =  1.000e+02

        Result of Checking:

        x1  = 1.001e+07  y1  = 1.001e+07  z1  = 2.002e+07
        vx1 = -9.990e+03 vy1 = 9.990e+03  vz1 = 1.998e+04
```

Figure 6.4 : Result for a Different Initial Guess (4/8)

```
    ****************************

    For the sample points:

    x[ 6] =  1.000e+02
    x[ 7] = -1.000e+02
    x[ 8] = -1.000e+02

Iteration

    0          X  = -2.10e+07     Y  =  0.00e+00     Z  =  0.00e+00
               VX =  0.00e+00     VY =  0.00e+00     VZ =  0.00e+00
               Error =  3.82e+07

    1          X  =  1.00e+07     Y  =  1.00e+07     Z  =  2.00e+07
               VX = -1.01e+04     VY =  1.01e+04     VZ =  2.01e+04
               Error =  9.67e+01

    2          X  =  1.00e+07     Y  =  1.00e+07     Z  =  2.00e+07
               VX = -1.01e+04     VY =  1.01e+04     VZ =  2.01e+04
               Error =  9.21e-10


        The solution is:

        x[ 0] =  1.002e+07
        x[ 1] =  1.000e+07
        x[ 2] =  2.000e+07
        x[ 3] = -1.008e+04
        x[ 4] =  1.010e+04
        x[ 5] =  2.010e+04
        x[ 6] =  1.000e+02
        x[ 7] = -1.000e+02
        x[ 8] = -1.000e+02

        Result of Checking:

        x1  = 1.001e+07  y1  = 1.001e+07  z1  = 2.002e+07
        vx1 = -9.990e+03 vy1 = 9.990e+03  vz1 = 1.998e+04
```

Figure 6.4 : Result for a Different Initial Guess (5/8)

```
    ****************************

    For the sample points:

    x[ 6] =  1.000e+02
    x[ 7] = -1.000e+02
    x[ 8] =  1.000e+02

Iteration

    0           X  = -2.10e+07      Y  =  0.00e+00      Z  =  0.00e+00
                VX =  0.00e+00      VY =  0.00e+00      VZ =  0.00e+00
                Error =  3.82e+07

    1           X  =  1.00e+07      Y  =  1.00e+07      Z  =  2.00e+07
                VX = -1.01e+04      VY =  1.01e+04      VZ =  1.99e+04
                Error =  9.67e+01

    2           X  =  1.00e+07      Y  =  1.00e+07      Z  =  2.00e+07
                VX = -1.01e+04      VY =  1.01e+04      VZ =  1.99e+04
                Error =  9.22e-10

        The solution is:

        x[ 0] =  1.002e+07
        x[ 1] =  1.000e+07
        x[ 2] =  2.000e+07
        x[ 3] = -1.008e+04
        x[ 4] =  1.010e+04
        x[ 5] =  1.990e+04
        x[ 6] =  1.000e+02
        x[ 7] = -1.000e+02
        x[ 8] =  1.000e+02

        Result of Checking:

        x1  = 1.001e+07  y1  = 1.001e+07  z1  = 2.002e+07
        vx1 = -9.990e+03 vy1 = 9.990e+03  vz1 = 1.998e+04
```

Figure 6.4 : Result for a Different Initial Guess (6/8)

```
        ****************************

        For the sample points:

        x[ 6] =  1.000e+02
        x[ 7] =  1.000e+02
        x[ 8] = -1.000e+02

Iteration

    0           X  = -2.10e+07     Y  =  0.00e+00     Z  =  0.00e+00
                VX =  0.00e+00     VY =  0.00e+00     VZ =  0.00e+00
                Error =  3.82e+07

    1           X  =  1.00e+07     Y  =  1.00e+07     Z  =  2.00e+07
                VX = -1.01e+04     VY =  9.90e+03     VZ =  2.01e+04
                Error =  9.67e+01

    2           X  =  1.00e+07     Y  =  1.00e+07     Z  =  2.00e+07
                VX = -1.01e+04     VY =  9.90e+03     VZ =  2.01e+04
                Error =  9.21e-10


            The solution is:

            x[ 0] =  1.002e+07
            x[ 1] =  1.000e+07
            x[ 2] =  2.000e+07
            x[ 3] = -1.008e+04
            x[ 4] =  9.900e+03
            x[ 5] =  2.010e+04
            x[ 6] =  1.000e+02
            x[ 7] =  1.000e+02
            x[ 8] = -1.000e+02

            Result of Checking:

            x1  = 1.001e+07  y1  = 1.001e+07  z1  = 2.002e+07
            vx1 = -9.990e+03 vy1 = 9.990e+03  vz1 = 1.998e+04
```

Figure 6.4 : Result for a Different Initial Guess (7/8)

```
     ****************************

     For the sample points:

     x[ 6] =  1.000e+02
     x[ 7] =  1.000e+02
     x[ 8] =  1.000e+02

Iteration

  0           X  = -2.10e+07      Y  =  0.00e+00      Z  =  0.00e+00
              VX =  0.00e+00      VY =  0.00e+00      VZ =  0.00e+00
              Error =  3.82e+07

  1           X  =  1.00e+07      Y  =  1.00e+07      Z  =  2.00e+07
              VX = -1.01e+04      VY =  9.90e+03      VZ =  1.99e+04
              Error =  9.67e+01

  2           X  =  1.00e+07      Y  =  1.00e+07      Z  =  2.00e+07
              VX = -1.01e+04      VY =  9.90e+03      VZ =  1.99e+04
              Error =  9.20e-10


         The solution is:

         x[ 0] =  1.002e+07
         x[ 1] =  1.000e+07
         x[ 2] =  2.000e+07
         x[ 3] = -1.008e+04
         x[ 4] =  9.900e+03
         x[ 5] =  1.990e+04
         x[ 6] =  1.000e+02
         x[ 7] =  1.000e+02
         x[ 8] =  1.000e+02

         Result of Checking:

         x1  = 1.001e+07  y1  = 1.001e+07  z1  = 2.002e+07
         vx1 = -9.990e+03 vy1 = 9.990e+03  vz1 = 1.998e+04
```

Figure 6.4 : Result for a Different Initial Guess (8/8)

7. Conclusion

The Titan II Space Launch Vehicle Digital Flight Control System is modeled using the Dynamic Flowgraph Methodology. This DFM model of the embedded system can be used to analyze the system to discover possible failures. This is an attempt to test DFM on a real embedded system in which the software code is available.

Dynamic Flowgraph Methodology is a tool for analyzing embedded systems in a "systems" approach. This tool is developed to fill in the deficiencies found in currently available reliability and safety analysis approaches. These existing approaches generally follow the philosophy of separating the hardware and software porions in the assurance analysis. On the other hand, DFM is a tool that integrates in one process the modeling and analysis of hardware and software components of an embedded system. In addition, depending on the availability of the software code used in the embedded system, DFM can focus the analysis on the software design or the software implementation. This paper demonstrates the latter situation.

The modeling framework in DFM is capable of representing features of modern day embedded systems. In particular. DFM allows functional, discontinuous, and dynamic features to be represented in the model. For the Titan II embedded system, the functional behavior is modeled by process variable nodes, transfer boxes, and causality edges, the discrete behavior is represented by process variable nodes and conditioning edges, while the software executions are modeled using the time-transition network. The model developed will then be used in Step 2 for identifying unwanted behavior. The algorithmic approach of backtracking and the knowledge base established in Step 1 can allow Step 2 to be automated. This is very important for analyzing a complex system such as the Titan II embedded system.

One of the steps in developing the DFM model of the Titan II embedded system is the construction of decision tables. These decision tables represent relationships between various states of different parameters in the model. In constructing the decision tables for subroutines in the Titan II flight control software, the problem of combinatorial explosion arises. As each subroutine takes in many input parameters, a huge number of combinations of these input parameters has to be sampled for each subroutine in order to complete the entries in the decision table. In addition, the size of the decision table makes it very time consuming to look up the entries during backtracking of the model.

As the subroutines in the Titan II flight control software implement equations with distinct physical meaning, it is possible to take advantage of this fact and avoid the construction of the decision tables prior to the analysis. In backtracking the DFM model to construct a timed-fault tree, if a subroutine is encountered and certain output values calculated from this subroutine exist in a branch in the timed fault tree, this subroutine can be solved in reverse to enter the entries next level down the fault tree. 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.

8. References

Beizer, B., (1990), *Software Testing Techniques*, Van Nostrand Reinhold.

Caldarola, L., (1980), Fault Tree Analysis with Multistate Components, *Synthesis and Analysis Methods for Safety and reliability Studies*, G.E. Apostolakis, S. Garribba, G. Volta, Eds. New York: Plenum Press, pp. 199-248.

Dummer, G.W.A., Reiche, H. and Hura, G.S., (1991), Special Issue: Petri Nets and Related Graph Models, *Microelectronics and Reliability*, **31**.

Fröberg, C.E., (1985), *Numerical Mathematics: Theory and Computer Application*, Benjamin/Cummings.

Garrett, C., (1993), *The Dynamic Flowgraph Methodology: A Methodology for Assessing the Dependability of Embedded Software Systems, M.S. thesis*, University of California, Los Angeles.

Guarro, S.B. and Okrent, D., (1984), The Logic Flowgraph: A New Approach to Process Failure Modeling and Diagnosis for Disturbance Analysis Applications, *Nuclear Technology*, **67**, pp. 348-359.

Guarro, S.B., (1988), PROGRAF_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*, March.

Guarro, S.B., (1990), Diagnostic Models for Engineering Process Management: A Critical Review of Objectives, Constraints and Applicable Tools, *Reliability Engineering and System Safety*, **30**, pp. 21-50.

Harvey, P.R., (1982), *Fault-Tree Analysis of Software, M.S. Thesis*, University of California, Irvine.

Henley, E.J. and Kumamoto, H., (1991), *Probabilistic Risk Assessment: Reliability Engineering, Design, and Analysis*, IEEE Press.

Johnson, L.W. and Riess, R.D., (1982), *Numerical Analysis*, Addison-Wesley.

Lapp, S.A. and Powers, G.J., (1977), Computer-aided Synthesis of Fault-trees, *IEEE Transaction on Reliability*, **R-26**, pp. 2-13.

Leveson, N.G. and Harvey, P.R., (1983), Analyzing Software Safety, *IEEE Transaction on Software Engineering*, **SE-9**, pp. 569-579.

Leveson, N.G. and Stolzy, J.L., (1987), Safety Analysis Using Petri Nets, *IEEE Transaction on Software Engineering*, **SE-13**, pp. 358-363.

Maron, M.J., (1987), *Numerical Analysis: A Practical Approach*, MacMillan.

Martin Marietta Astronautics, (1988), *Guidance, Control, and Ground Equations for Flight Plan XX Volume II: Flight Control Equations XX-T001-II-08*, June 24.

Martin Marietta Astronautics, (1991), *Guidance, Control, and Ground Equations for Flight Plan XX Volume I: Guidance Equations XX-U001-I-05*, February 18.

Morgan, E.T. and Razouk, R.R., (1987), Interactive State-Space Analysis of Concurrent Systems, *IEEE Transactions on Software Engineering*, **SE-13**, pp. 1080-1091.

Murata, T., (1989), Petri Nets: Properties, Analysis and Applications, *Proceedings of the IEEE*, **77**, pp. 541-580.

Muthukumar, C.T., Guarro, S.B. and Apostolakis, G.E., (1991), Logic Flowgraph Methodology: A Tool for Modeling Embedded Systems, *Proceedings of the IEEE/AIAA 10th Digital Avionics Systems Conference*, Los Angeles, CA., Oct. 14-17, pp.103-109.

Narayana, K.T. and Aby, A.A., (1988), Specification of Real-Time Systems in Real-Time Temporal Interval Logic, *Proceedings of the 1988 Conference on Real-Time Systems,* IEEE Press.

Peterson, J.L., (1981), *Petri Net Theory and the Modeling of Systems*, Prentice-Hall.

Razouk, R.R. and Gorlick, M.M., (1989), A Real-Time Interval Logic for Reasoning About Executions of Real-Time Programs, *Proceedings of ACM SIGSOFT '89*, ACM Press Software Engineering Notes, **14**, pp. 10-19.

Salem, S.L., Apostolakis, G.E. and Okrent, D., (1977), A New Methodology for the Computer-Aided Construction of Fault Trees, *Annals of Nuclear Energy*, **4**, pp. 417-433.

Salem, S.L., Wu, J.S. and Apostolakis, G.E., (1979), Decision Table Development and Application to the Construction of Fault Trees, *Nuclear Technology*, **42**, pp. 51-64.

Appendix A Logic Flowgraph Methodology (LFM)

A.1 LFM Concept

The Logic Flowgraph Methodology (LFM) [Guarro and Okrent, 1984], [Guarro, 1988], [Guarro, 1990] was originally developed as a method for analyzing/diagnosing plant processes with feedback and feedforward control loops. The LFM models take the form of directed graphs, with relations of causality and conditional switching actions represented by "causality edges" and "conditional edges" that connect network nodes and special operators. Of these, causality edges represent important process variables and parameters, and conditional 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.

The application of LFM, like that of its derivative, DFM, is typically a two-step process:

Step 1: The construction of a model for the system of interest. This model is built by first identifying all of the basic process parameters by which the system behavior can best be described and then by expressing the fundamental cause-effect interactions (and the conditioning effects on these interactions effected by faults and operational mode changes) among the parameters.

Step 2: The search for the manner in which specific process states (identified by the values that certain parameters may assume within the given process) may occur as the result of the propagation through the system of perturbations produced by basic root cause events (such as system component faults or manifestations of process-control logic errors). This information is ordered in the form of a fault tree using AND and OR gates. This stage is executable in the form of an automated procedure that traces cause-effect relationships backwards through the LFM network model.

A.2 Example

The application of LFM to a simple hardware system is illustrated in Figure. A.1, where a valve is used to control the flowrate downstream. In Figure. A.1(a), a piping and instrumentation diagram (P&ID) is drawn to describe the functional layout of the system, its components, and other elements of basic engineering data regarding the process. Other important attributes, most notably the ones linked to operational logic and control modes as well as the analyst's own understanding of the system, while not directly contained nor implicitly expressed in the P&ID, are nevertheless represented in the LFM model of the system (Figure. A.1(b)). The LFM model is built with physical parameters UP, F, FM, and VX as continuous variable nodes, where

UP = Upstream pressure,
F = Flow rate,
FM = Flow rate measured,
VX = Valve position at the present time,

and SF and CF as discrete variable nodes, where:

SF = Sensor state,
CF = Control Function.

The relationships between parameters are represented by gains in transfer boxes, which may be different for different conditions. Edges connect nodes through transfer boxes. An example of how relationships are represented in the LFM model is the direct proportionality relationship between nodes UP and F. This is represented by a "/" in the transfer box between the nodes. The two nodes are connected through the transfer box using directed edges (Figure. A.1(b)). According to the different degraded states of the sensor (SF), the relationship between the nodes F and FM may change. This is clearly shown in the model (Figure. A.1(b)).

It should be noted that the results of an LFM analysis are obtained in the form of fault trees, which show how the investigated system/process states may occur. LFM thus shares, in the final form of the results it provides, many of the features of fault tree analysis. The difference, however, is that it provides a documented model of the system's behavior and interactions, which fault tree analysis does not provide directly. The most important feature of this methodology is that once an LFM model has been developed, it is not necessary to construct separate models for each system state of interest (as is the case in fault tree analysis).

In Figure. A.1(c), the fault tree for the top event, "flow rate is high," is derived. By working backward through the LFM model starting from the flow rate node F, we can determine that the state, "flow rate F is high," is caused by either "upstream pressure UP is high" AND "the valve opening is nominal," OR "the valve is completely open." This information is implicitly contained in the LFM input operator before the node F. Underlying this input operator is a decision table constructed by determining the states of F from the combinations of the states of UP and VX. Thus, given a particular state of F, the information organized in the decision table can be used in reverse to determine the combinations of UP and VX which cause this particular state of F. This information is explicitly denoted in the resulting fault tree by connecting events with logical AND and OR gates.

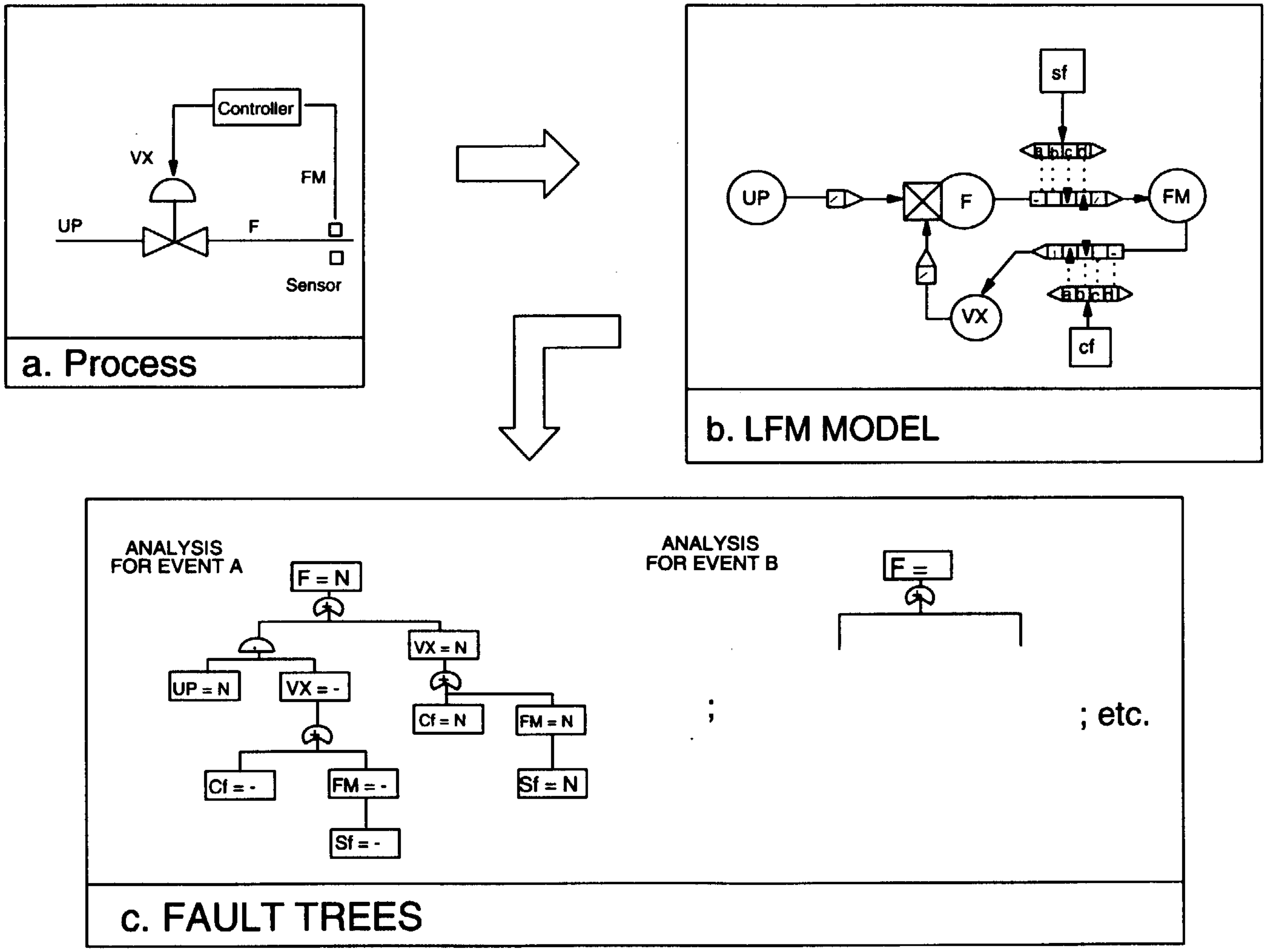


Figure A.1 : Example of LFM Model and Fault Analysis

Appendix B Dynamic Flowgraph Methodology Analysis

B.1 Framework for Implementing Step 2

The result of the analysis of the digital control system model developed in Step 1 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. Essentially, a timed fault tree is like a series of snapshots of the system evolution, with each snapshot presented as a conventional fault tree.

Conventional fault tree analysis is very well established in the areas of safety and reliability analysis. Originally developed at the Bell Laboratory, fault tree analysis has been used to analyze nuclear power plants [Henley and Kumamoto, 1991], chemical processes [Lapp and Powers, 1977], and software [Leveson and Harvey, 1983]. It should be noted at this point 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-state logic fault trees [Caldarola, 1980].

B.2 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 model 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.

B.3 Example Application

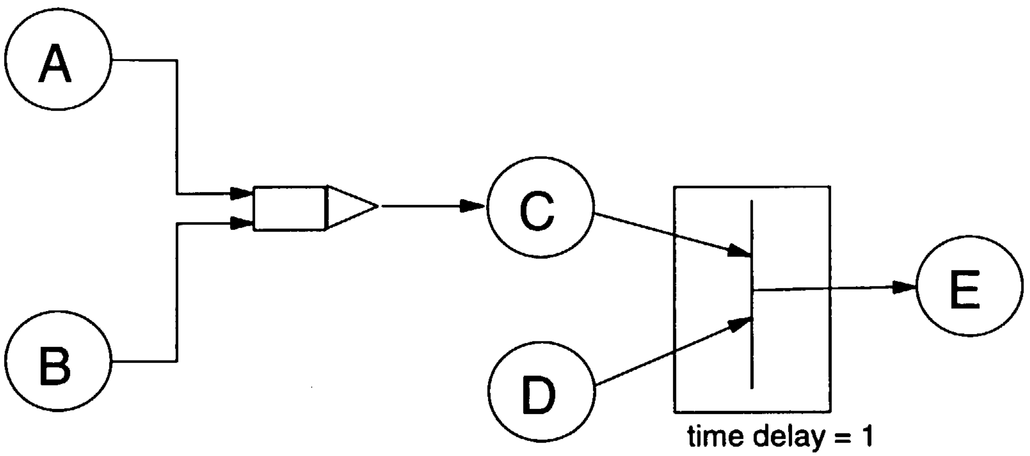
An example of how to construct a timed fault tree from the information provided in the

DFM model is shown in Figure. B.1. The hypothetical system model consists of 5 Process variable nodes representing variables A, B, C, D, and E, with each variable discretized into 3 states (0, 1, 2). Nodes A and B are inputs to node C through a transfer box, and nodes C and D are inputs to node E through a transition box. The decision tables are also shown in Figure. B.1.

Suppose we want to analyze how E = 2 is reached. This is done by constructing a timed fault tree with E = 2 as the top event. The timed fault tree is shown in Figure. B.2.

From decision table II in Figure. B.1, we can determine that E = 2 is caused by either C = 0 and D = 1, or C = 2 and D = 0. This information is represented in the first level below E = 2. Note that C, D, and E are linked by a transition box, so in the timed fault tree we have indicated that there is a time transition between satisfying the states of C and D, and reaching the state E = 2. Next, we find from decision table I in Figure. B.2 that C = 0 is due to A = 0 and B = 1, and C = 2 is caused by A = 1 and B = 2, or A = 2 and B = 0. This information is then represented in the timed fault tree at the next level.

The example just given is simple enough to illustrate how events can be traced to their cause through transfer boxes and transition boxes. In real digital control systems, there are usually feedback or feedforward characteristics. This can cause a node to 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.



A	B	C
0	0	1
0	1	0
0	2	1
1	0	1
1	1	1
1	2	2
2	0	2
2	1	1
2	2	1

Decision Table I

C	D	E
0	0	0
0	1	2
0	2	1
1	0	0
1	1	0
1	2	1
2	0	2
2	1	1
2	2	0

Decision Table II

Figure B.1 : Illustration of Fault Tree Construction

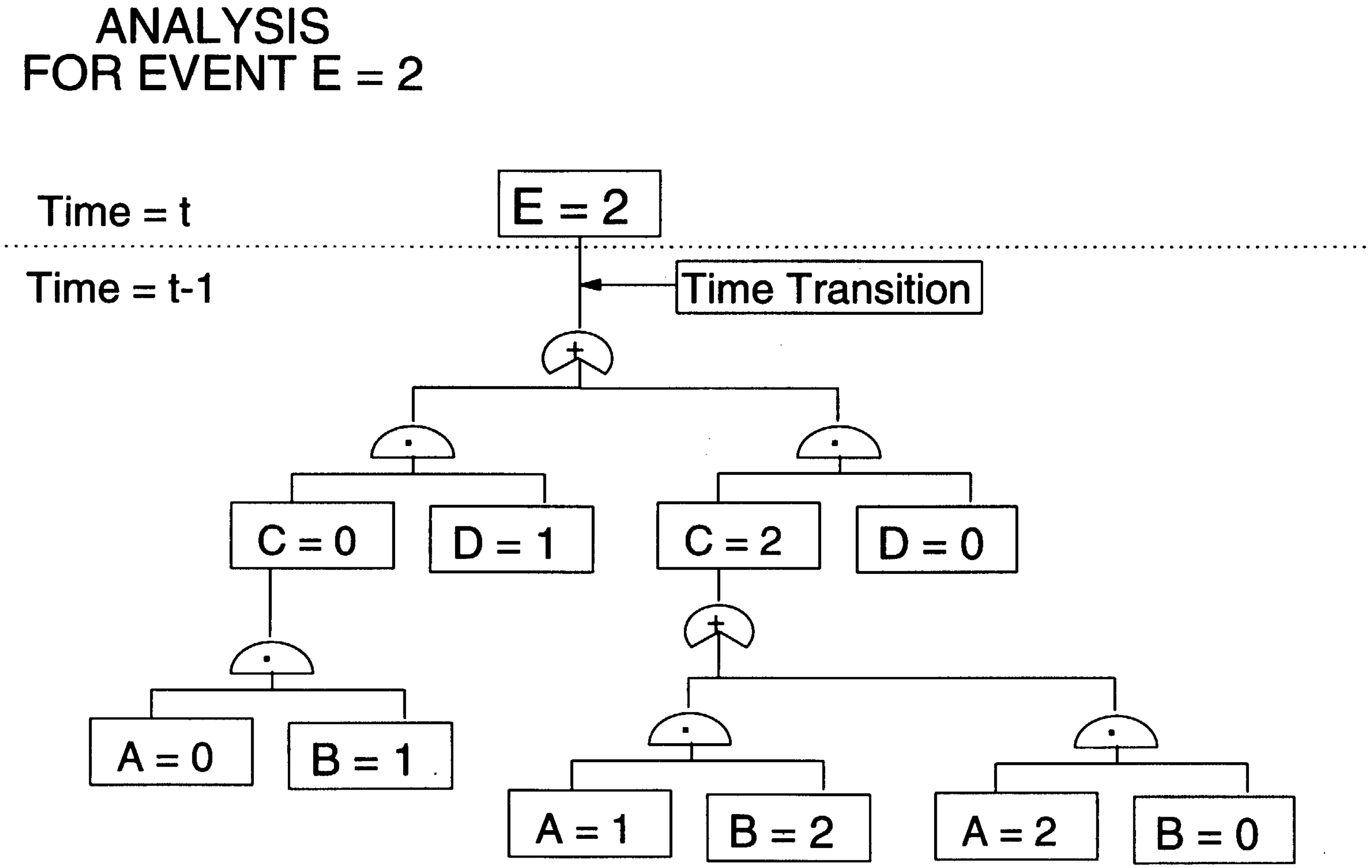


Figure B.2 : A Timed Fault Tree for the DFM Model in Figure B.1

Appendix C Titan II Space Launch Vehicle Digital Flight Control System

The function of the Titan II SLV Digital Flight Control System [Martin Marietta, 1988], [Martin Marietta, 1991] is to maintain attitude stability of the vehicle from liftoff to payload release. The system measures the orientation and acceleration and uses these measurements to calculate the appropriate engine deflection to maintain flight stability.

The Titan II SLV DFCS consists of the Missile Guidance Computer and the flight control software, the Attitude Rate Sensing System, the Inertial Measurement Unit (IMU), and the hydraulic actuators. Each of these subsystems will be discussed below.

C.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 routine is composed of a number of subroutines for carrying out dedicated calculations.

The Real-Time-Interrupt is responsible for reading in data from the sensing devices, i.e. from the IMU and the rate sensing system, handling telemetry, giving commands to the hydraulic actuators, and issuing engine shutdown commands. Inputs are read in and outputs are given out every 40 msec (during powered flight) or 20 msec (during 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 certain phase.

The three major routines share the computer resource in the Missile Guidance Computer.

The sharing of the computer resources is shown in Figure C.1. The RTI has the highest priority, it is executed every 5 msec 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 msec during powered flight and every 20 msec during coast flight. The actual calculation time is less than 40 msec/ 20 msec. 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 sec. Calculation takes place in the time-slot unused every Minor Cycle. All the major cycle calculation actually completes in less than 1 sec. The time-slot normally used by the Major Cycle after its completion is dedicated to background calculations. Background calculation is not relevant to flight control and will not be discussed here.

C.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 schematic of a gyro is shown in Figure C.2. It 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 $\mathbf{O} = \mathbf{H} \times \omega$. Thus, the amount of angular motion can be determined by measuring the rotation about the output axis.

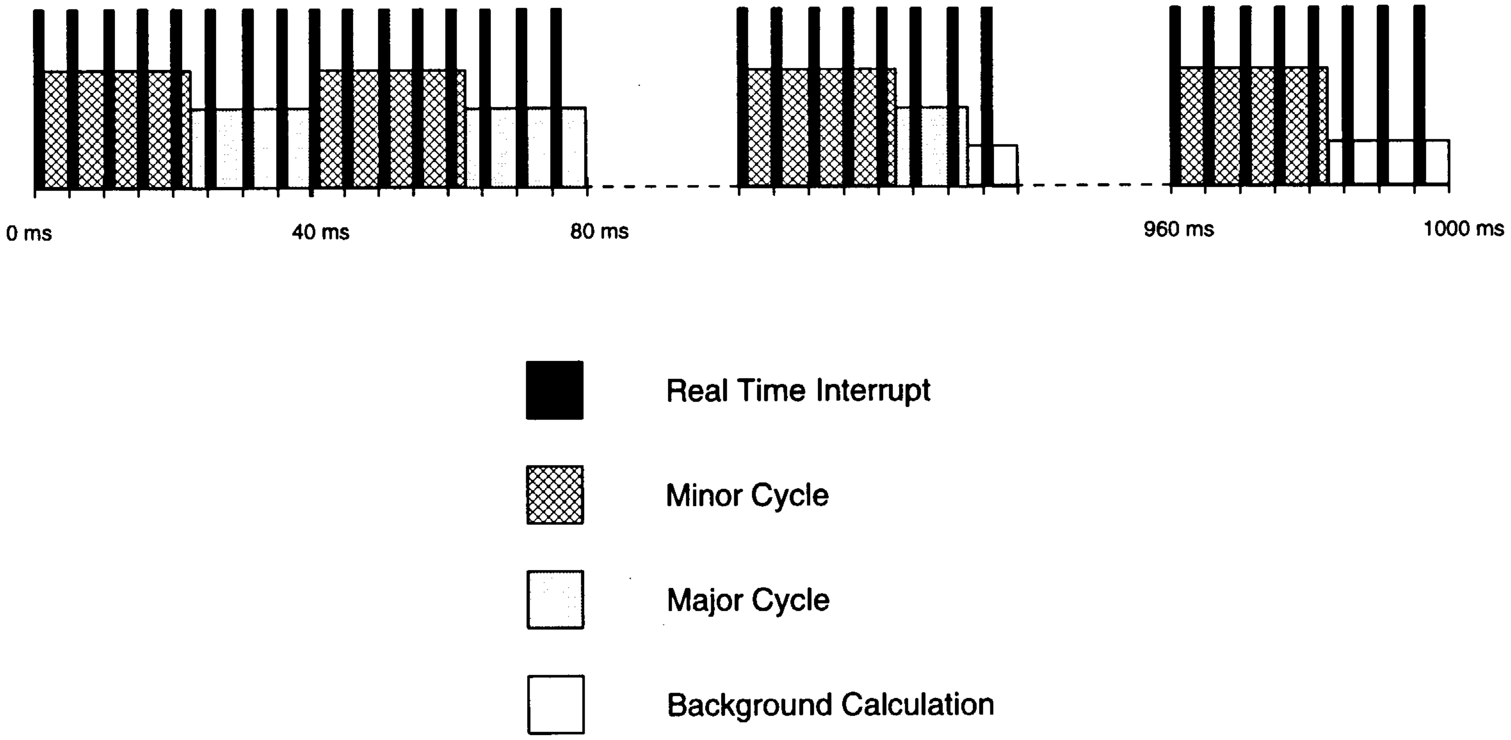


Figure C.1 : Sharing of Computer Resources

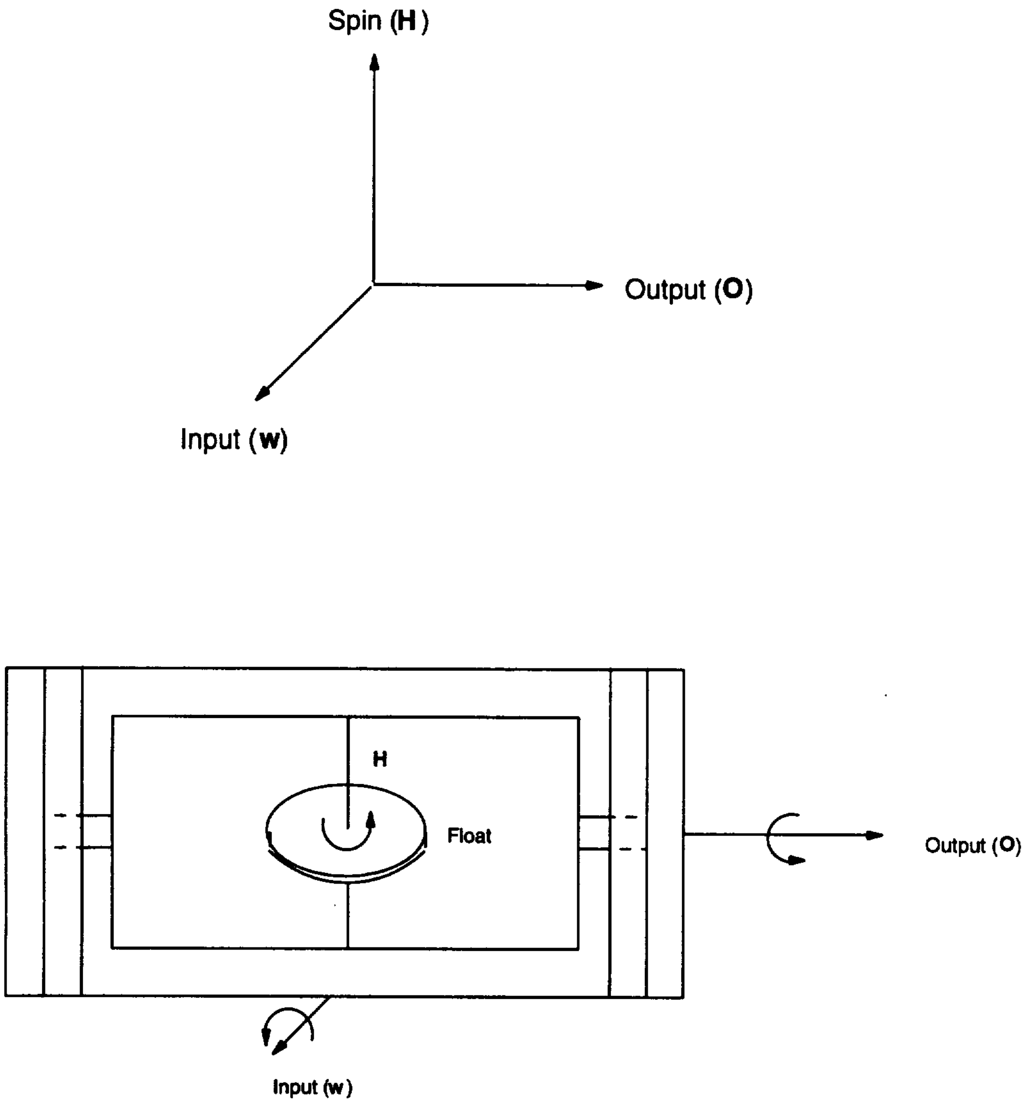


Figure C.2 : Schematic of a Gyro

C.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 IMU is shown in figure C.3.

A schematic of an accelerometer is shown in Figure C.4. It consists of a shuttle mass located inside a case. When the case undergoes an acceleration **a**, the shuttle mass moves against the spring due to its inertia. The 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 a an angular motion θ into a two channel electrical output signal. One channel gives K $\sin(\theta-120^\circ)$ and the other gives K $\sin(\theta-60^\circ)$, where K is the instrument's constant.

C.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°. In Stage I, where there is two thrust chambers, the schematic for roll, pitch, and yaw corrections are shown in Figure C.5.

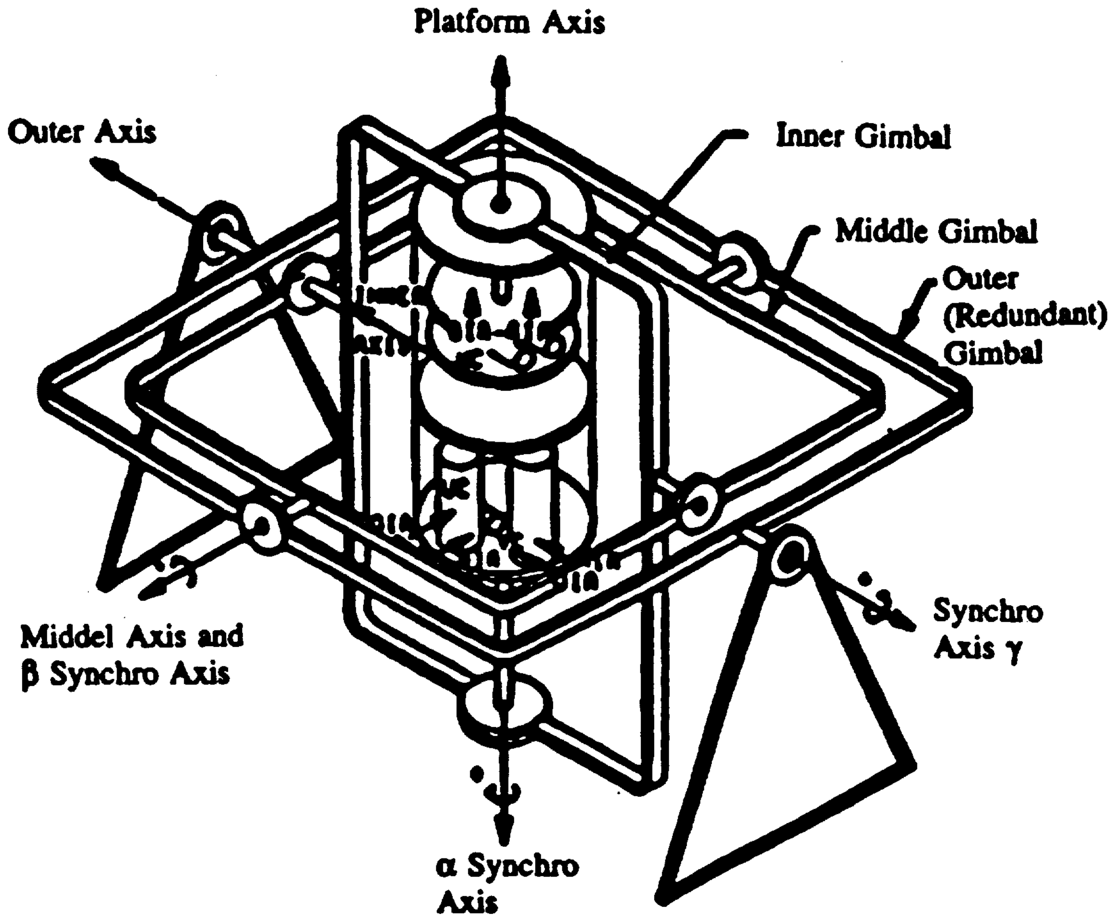


Figure C.3 : The Inertial Measurement Unit

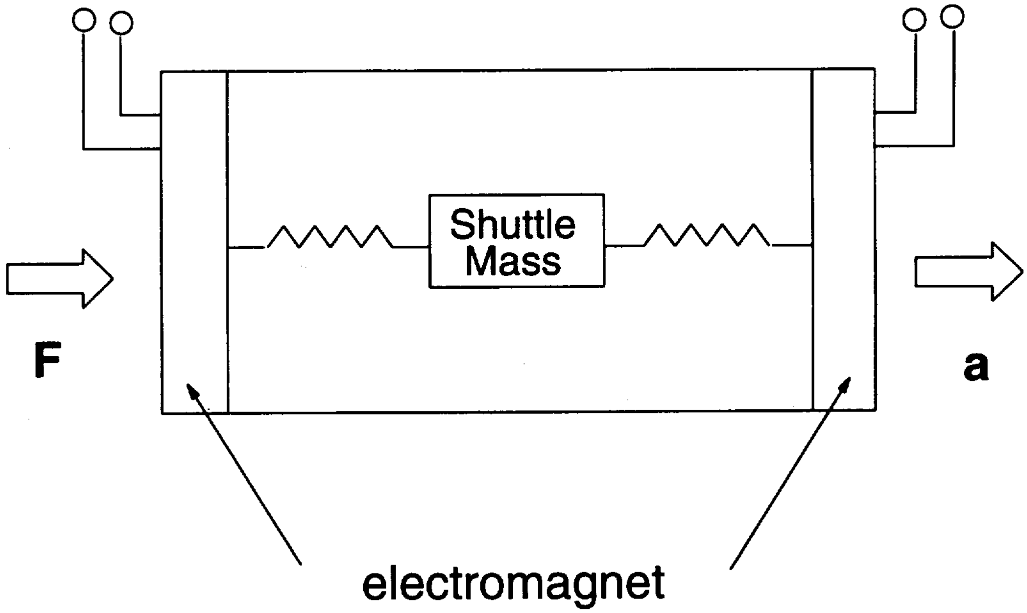


Figure C.4 : Schematic of an Accelerometer

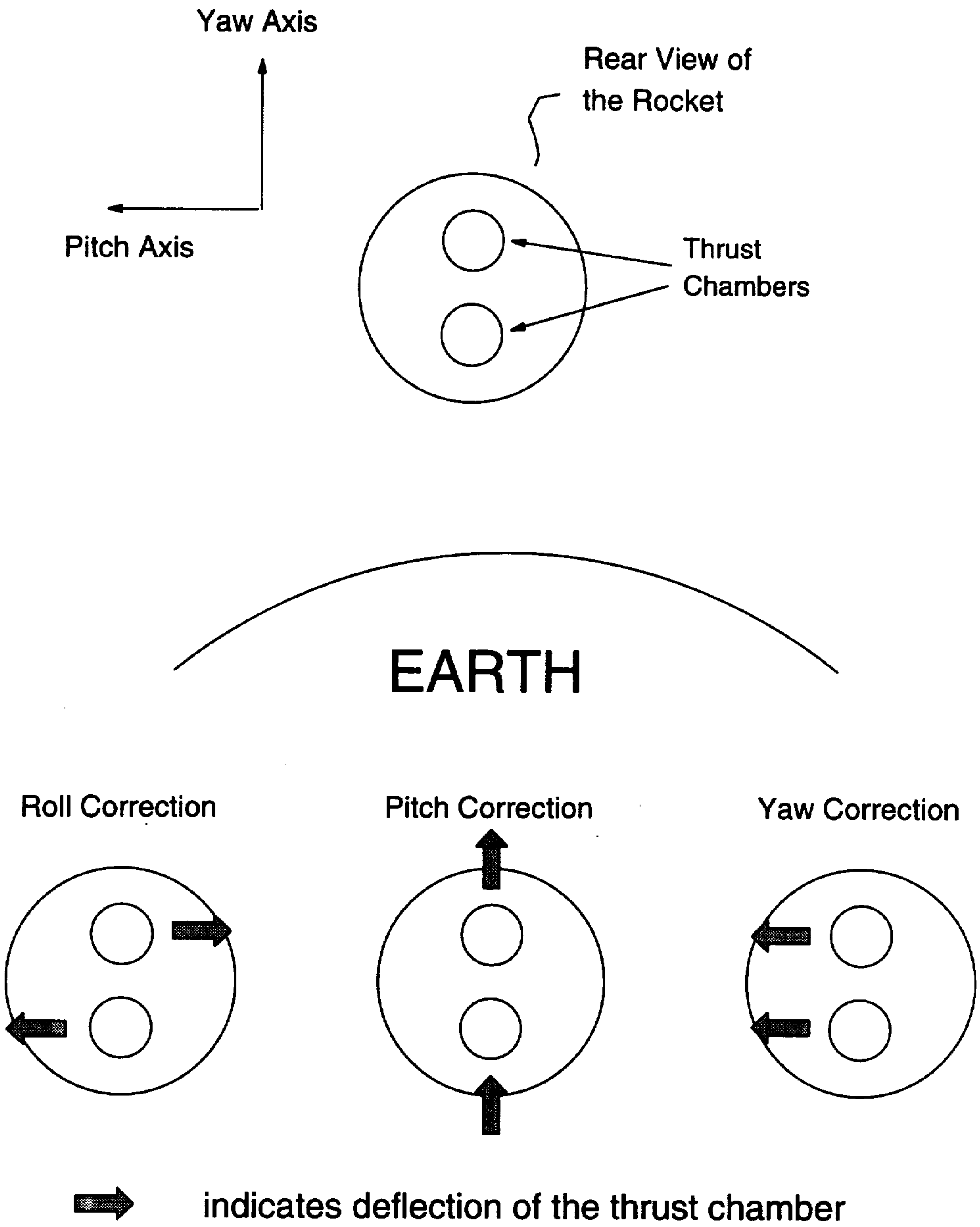


Figure C.5 : Working of the Hydraulic Actuators