Model based specification for developing safety-critical system-software

By

Anand K. Karasi

B. Tech. Aerospace Engineering
Indian Institute of Technology, Madras, 1997

M. Eng Aeronautics and Astronautics
Massachusetts Institute of Technology, Cambridge, MA, 1999

Submitted to the Department of Engineering Systems Division in partial fulfillment of the requirements for the degree of

Master of Science in Technology and Policy
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
May, 2000
[June 2000]

Signature of Author..... Signature redacted
Technology and Policy Program
Engineering Systems Division
May 5 , 2000

Certified by ... Signature redacted .....
Professor Nancy G. Leveson
Thesis Supervisor

Accepted by..... Signature redacted .....
Professor Daniel Hastings
Professor of Engineering Systems and Aeronautics and Astronautics
Chairman, Technology and Policy Program

Model based specification for developing safety-critical system-software

by

Anand K Karasi

Submitted to the Department of Engineering Systems Division on May, 2000 in partial fulfillment of the requirements for the degree of Master of Science in Technology and Policy

Abstract

Aerospace systems are complex reactive control systems, which exhibit sophisticated modes of behavior. With the advent of faster computers and availability of precise navigation sensors over the past few years, the degree of complexity has compounded. Software has been used to implement the numerous modes of behavior exhibited by these systems. In addition, many limitations of hardware are being overcome by implementing the desired behavior in software. Software is playing a more important in aerospace system design than before. Hence highest standards need to be applied during the design processes to ensure the quality of software. The current standards have been unsuccessful in containing software errors. Effective means to contain software errors have not been available until today. This has lead to a number of accidents caused due to software errors.

The short term as well of the long term implications of accidents on the industry and society have been considered in formulating policies. Current standards and regulations have failed to contain software related accidents due to their ambiguity and inability to address the root cause of these accidents – software is different, the way is designed and built; and the way it fails.

This thesis proposes the SpecTRM methodology as a means to incorporate safe design practices throughout the design processes for the development of safety-critical system-software. Requirements specifications for an autonomous helicopter were developed in the SpecTRM-Requirements Language. These specifications were converted into executable code and used in a closed loop simulation. The logic defined for the behavior of the helicopter, in the human readable specification, could be visually verified while a simulated autonomous mission was executed.

Thesis Supervisor: Nancy G. Leveson
Title: Professor, Department of Aeronautics and Astronautics, MIT

Acknowledgments

I would like to thank Prof. Nancy G. Leveson for introducing me to the field of aerospace software and guiding me in understanding the importance of safety and the reasons behind software errors. Jeffery Michael Thompson for patiently answering all my questions regarding Nimbus and helping me debug parts of the code remotely.

Christopher P Sanders was valuable in developing the interface between Nimbus and Draper FCS. I am thankful to the sponsors at Draper Laboratory for supporting the implementation of SpecTRM to their autonomous helicopter.

My lab-mates, Natasha and the M4 team (Mario, Maxime, Mark and Masa) for giving me feedback on some of the ideas mentioned in this thesis.

My sisters and father in India for proof reading this thesis over the Internet and e-mailing me their comments and corrections.

Table of Contents

List of Acronyms

DGPS	Differential Global Positioning System
FCS	Flight Control System
FMS	Flight Management System
GCU	Ground Control Unit
GPS	Global Positioning System
HOGC	Helicopter Onboard Guidance Controller
IMU	Inertial Measurement Unit
MPL	Mars Polar Lander
OBC	On Board Computer
RSML	Requirements State Machine Language
SpecTRM	Specification Tools and Requirements Methodology
SpecTRM-RL	Specification Tools and Requirements Methodology – Requirements Language
VB	Visual Basic

List of Figures

CHAPTER 1

Introduction

1.1 Background

The last century witnessed the dominance of "iron machines" (industries, trains, planes etc.) that marked the achievements of society. They defined the progress of mankind but at the cost of many accidents, which led to loss of lives, property and destruction of the environment. Today, at the dawn of the 21st century, these iron machines respond to the commands of "weightless bits" (software). Systems and the software required to operate them are getting more complex. The degree of complexity is at the verge of expanding beyond human capabilities. Currently, adequate tools are not available to assist in developing system requirements and architectures in the early stages of projects. Serious challenges are encountered in translating system requirements to software requirements. Many unsolved problems exist in changing and upgrading automated control tasks without introducing errors at the end of the life-cycle phase. In addition to the system engineering problems, these phases present the most serious and unsolved problems in validation and certification of the system.

A model-based software specifications method, called SpecTRM (Specification Tools and Requirements Methodology), has been proposed as an approach that can be used to develop safe software for safety-critical systems. This thesis shows that this model-based software specifications approach, involving state-of-the art requirements modeling and safety analysis techniques, can be used to develop safe software.

Advantages of using "model-based software specifications" will be highlighted while developing the specifications for an autonomous helicopter. Lessons learned from generating specifications for the autonomous helicopter will be used to formulate policies for certification and validation.

1.2 Motivation for the Thesis

Software requirements errors have been found to account for a majority of production software failures, and most serious accidents involving software relate to requirements errors, not coding or programming errors [14]. Aerospace companies have continually stated that their biggest cost driver for software are requirements errors and the lack of effective requirements analysis techniques to detect flawed requirements early in development. Writing and validating requirements for embedded systems is difficult. However they are an essential part of the systems engineering and certification of safety-critical complex systems.

Many accidents in the past can be attributed to software-related errors. Safety engineers understand those caused by non-software related errors fairly well and policy makers have developed regulations and standards to reduce such accidents. However, software-related mishaps have many unique characteristics, which are different from characteristics of traditional engineering. Accidents caused by errors in software are not very well understood. Hence, developing regulations to contain such accidents has been difficult. This has been exacerbated in the recent past with a surge in aerospace related accidents. Between mid-1998 and the end of 1999, an alarming number of mistakes and failures have disrupted America's military and civilian space programs[6]. United States

lost more rockets and satellites than in any comparable period since the early days of space exploration. Billions of dollars in payloads were lost, including important military surveillance and communication satellites. Both Government and industry initiated broad, independent reviews of the processes and procedures involved at all levels of launch vehicle design, production, assembly, and launch. Preliminary investigations into these catastrophes showed that they were caused by inadequate software debugging and poor attention to detail. Cost cutting has also been attributed to the significantly reduced U.S. capabilities to successfully launch and operate space missions [6]. Low project budgets and tight schedules have forced many management and engineering decisions, which compromise safety. Less thorough verification and validation processes have been adopted on unconventional and novel designs.

Software code verification by reviews and conventional methods like code walk-through have proven to be ineffective as seen by Mars Polar Lander and Ariane 5 failures. Failure analysis using logic flow diagrams to illustrate the logic failures showed that, instead of trying to understand the logic by reading the code, diagrams were more effective and provided more visibility for the walkthrough reviewers [25]. This motivated the investigation into the use of a dynamic executable logic specification as a method to review, validate and certify software code. Requirements specifications developed in SpecTRM-RL (Specification Tools and Requirements Methodology – Requirements Language) are executable.

1.3 Thesis Objectives and Outline

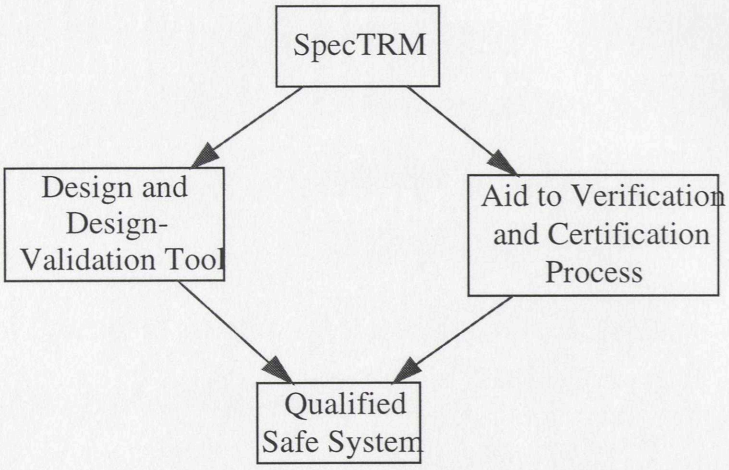


Fig 1.1 Spectrum as a Tool to Develop Safe System Software

This thesis describes the SpecTRM-RL specifications developed for an Autonomous Helicopter. The applicability and utility of this methodology for the design, development, validation and certification of safety-critical systems software has been investigated.

Three examples of recent accidents related to the aerospace industry have been studied to understand the cause and remedies proposed. These have been chosen from major sectors of the aerospace industry - Commercial (Boeing Sea launch), US Civilian (NASA) and Foreign (Aerospatiale) to eliminate problems that might be specific to particular industry sector. Chapter 2 discusses these accidents. The impact of such accidents on future projects is also assessed.

Investigations, their scope and recommendations, following these accidents are described in Chapter 3. It is hypothesized that these accidents are a result of ignoring the known problems of managing and engineering complex and large software systems. Chapter 3 provides a higher-level view of the causes of these accidents and attempts to abstract the root causes. Such a high-level view attempts to provide a perspective to avoid similar accidents in the future. In addition to the technical challenges involved in the design and development of safe software, the role of verification and validation processes and certification has been considered.

Chapter 4 provides a brief overview of SpecTRM methodology and its implementation in the Nimbus Environment. Chapter 5 describes the application of this method to the design of an executable specification for an autonomous helicopter controller.

The simulation set-up developed to implement a closed loop simulation is described in Chapter 6. Limitation of the current standards to prevent accidents has been discussed in Chapter 7. The SpecTRM methodology is suggested as a tool to certify safety critical software.

CHAPTER 2

Overview of accidents

A number of accidents over the last few decades related to software errors have been described by Leveson [11]. This chapter describes a few accidents in more recent times. Only the major accidents caused due to errors in software are mentioned to show the magnitude of losses and the impact on industry and society. These accidents show that improvements in system software development methods and processes need serious attention. Many software errors were detected by shear luck and catastrophic accidents were prevented. This shows that the potential of software errors to cause havoc is far greater that what has been witnessed in the last two years.

2.1 Accidents

Developing software for complex, reactive control systems is difficult. A surge in accidents in such systems was witnessed in the aerospace industry. Some of the most complex, innovative and expensive projects were destroyed due to errors in software. The following examples illustrate how software errors contributed to these accidents.

2.1.1 Sea Launch Failure

On march 12, 1999, the Sea Launch rocket fell into the ocean after liftoff from a floating platform. The pneumatic system, which is used to control a variety of systems on the booster, including the operation and actuation of a steering engine, failed to close a valve on the second stage of the Zenith 3SL booster. As a result, the pneumatic system

lost pressure before and after liftoff. By the time the second stage was started more than half of its pressure was lost, reducing the performance of the steering engine. This led to the booster drifting off course to the point where an automated self-destruct system on the booster was triggered.

Investigations have shown that a logic failure in ground support system software failed to properly configure the rocket before launch. This caused a valve in the second-stage pneumatic system to remain open when it should have been closed prior to launch. The rocket was carrying a $100 million satellite for ICO Global Communications. The exact details of the logic failure were not available at the time of writing this thesis. However it has been confirmed that the error lies in the software logic.

2.1.2 Ariane 5

On 4 June 1996, about 40 seconds after the initiation of the flight sequence, the Ariane 5 launch vehicle veered off its flight path, broke up and exploded. During the failure, the primary Inertial Reference System (SRI) shut down due to a software exception [9]. A software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating-point number, which was converted, had a value greater than what could be represented by a 16-bit signed integer. This resulted in an Operand Error. There were two SRIs operating in parallel, with identical hardware and software. One SRI was kept active and another on "hot" stand-by. If the on-board computer (OBC) detected that the active SRI has failed it immediately switches to the other one. In Ariane 5 the stand-by SRI had shut down before the active SRI for the same reason. This resulted in the on-board computer

sending a diagnostic pattern to the booster nozzles. The nozzles responded by going into full deflection, thus forcing the vehicle to exceed its maximum angle of attack. The problem was that the inertial reference system used on Ariane 5 was the same one used on Ariane 4, which had totally different requirements.

The error occurred in a part of the software that performs alignment of the strap-down inertial platform. This software module computes meaningful results only before lift-off. As soon as the launcher lifts off, this function serves no purpose. Software was considered to be correct until it is shown to be at fault [9]. The success of the software on Ariane 4 had strengthened that argument. The failure of the maiden flight caused delays in delivering payloads on the next Ariane 5 launcher. The cost of the launch vehicle was estimated at about $1 billion.

2.1.3 Mars Polar Lander

On December 3rd, 1999, about ten minutes before it was expected to land on the south polar region of Mars, the Mars Polar Lander (MPL) lost contact with Earth and it was never regained. It's assumed that MPL crashed on the Martian surface. The probable cause of the loss of MPL was identified as a premature trigger of a touchdown sensor. It was discovered that the software did not behave in the manner intended [25, section 7.7.2]. The descent engines are shut down by a command initiated by the flight software. This command is triggered when two conditions are satisfied – variable "Indicator State" is set to True and variable "Event Enabled" is set to Enabled. The "Event Enabled" variable is set to Enabled at all altitudes less than 40 m. This was due to the limitation in the resolution of the measuring device. The " Indicator State" is set to

True when the first landing leg senses touchdown. If the touchdown sensor in that leg fails to detect the touchdown, the second leg to touch down will trigger the engine shutdown. This logic is intended to prevent the Lander from tipping over when it has a skewed attitude relative to the surface during touchdown.

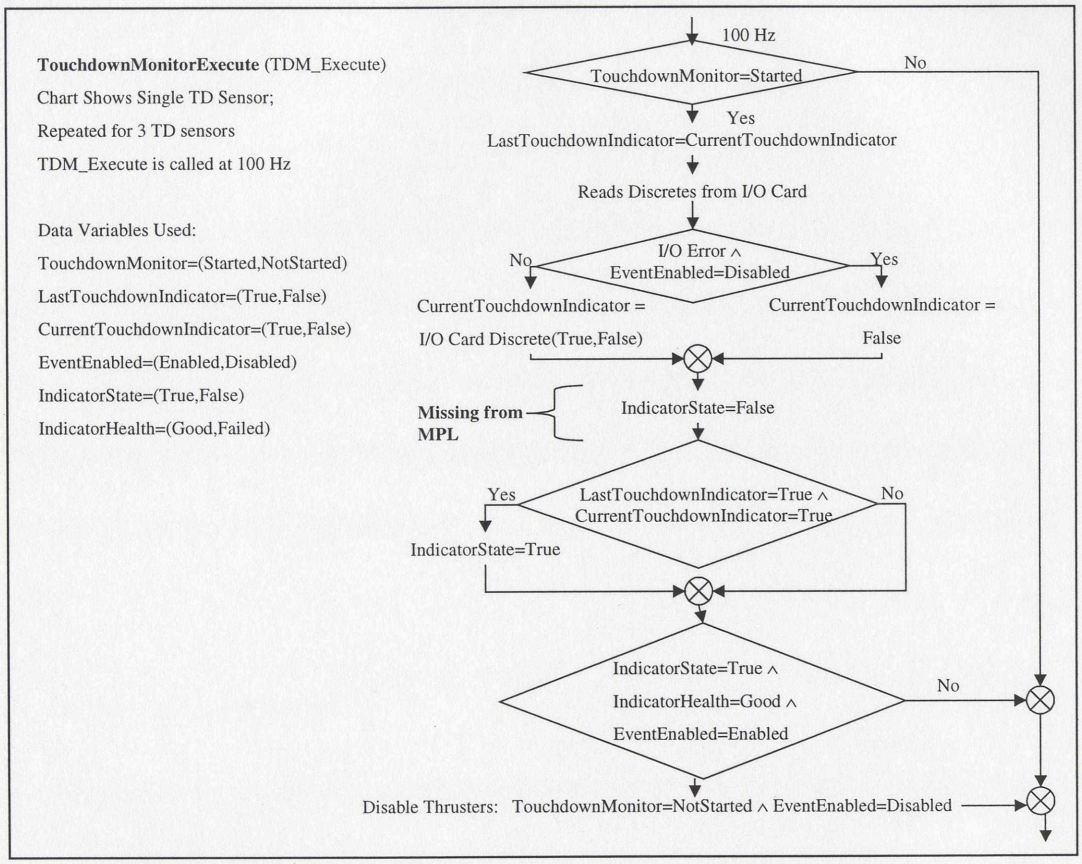


Fig 2.1 Error in Software Logic of Mars Polar Lander

A spurious signal was generated by one of the touchdown sensors at an altitude of 1500 m. This resulted in the "Indicator State" being set to True. This indicator was not reset to False because the anticipated spurious signal was not taken into account in the design of the MPL software. At 40 m altitude, the thruster shutdown trigger depended solely upon the value of the "Indicator State". As it was already set to True the thruster

was shut down and the spacecraft crashed on to the surface. A very important and expensive spacecraft was lost as a result of a logic error and incomplete specification.

The software included a design error that was not detected in the software walkthrough process or discovered in subsequent testing of the software. The design error was discovered after a fault-tree analysis led to the examination of the code and the preparation of code descriptions for reviews by outside reviewers.

The reasons mentioned in the above examples are only a few that can potentially go wrong in any complex systems project. And these errors are often difficult to detect. Evidence of the cause of the failure is often destroyed with the system. In spite of thorough, long and expensive investigations, often the reasons are only speculative as was evident from the MPL failure.

2.2 Impact of Accidents

The loss from accidents in the aerospace industry is not limited to the cost of the vehicle or the payload alone. These costs are usually recoverable through insurance. The accidents impact the industry and the society on a much larger scale.

Accident investigations are initiated immediately following the accident. These involve review boards with experts from different fields. Until the recommendations are made, work on future projects are delayed. After the faultfinding process is completed and the errors identified, a return-to-flight program is initiated. All these events delay launch of future projects. Recommendations often suggest the use of proven

technologies and methods to reduce the possibility of accidents. Such recommendations restrict the ability to extend the envelope and try innovative projects.

The investigations would be more intensive if accidents involve loss of human life [26]. The recommendations might suggest delaying future missions indefinitely until the safety of the project is assured or force the project to be terminated.

Due to the magnitude of size and costs involved in aerospace projects, many private ventures today involve partnerships that share the capital investment and development of parts of the project. Software errors causing accidents do not leave clear evidence about the nature of the error. This requires extensive investigation often involving long periods of time. Since software controls almost all subsystems, each of the partners is forced to make an investigation into their contributions to the project. All four companies involved with Sea Launch - Boeing, Energia, Yuzhnoye, and Norwegian shipbuilder Kvaerner Group - are conducting independent investigations into the cause of the Sea Launch failure. During the investigations, with no sufficient proof of the failure, the partners end up blaming each other. The blame is often placed on the most obvious cause. For example, in the case of the sea launch failure, it was believed that the Energia-provided guidance computer caused a failure, an explanation Energia rejected. Such allegations and rebuttals strain the relation between partners and hinder smooth operations.

Fig 2.2 shows how the faster-better-cheaper initiative results in a lower quality product that eventually leads to project failure. Project failure in turn result in delays in

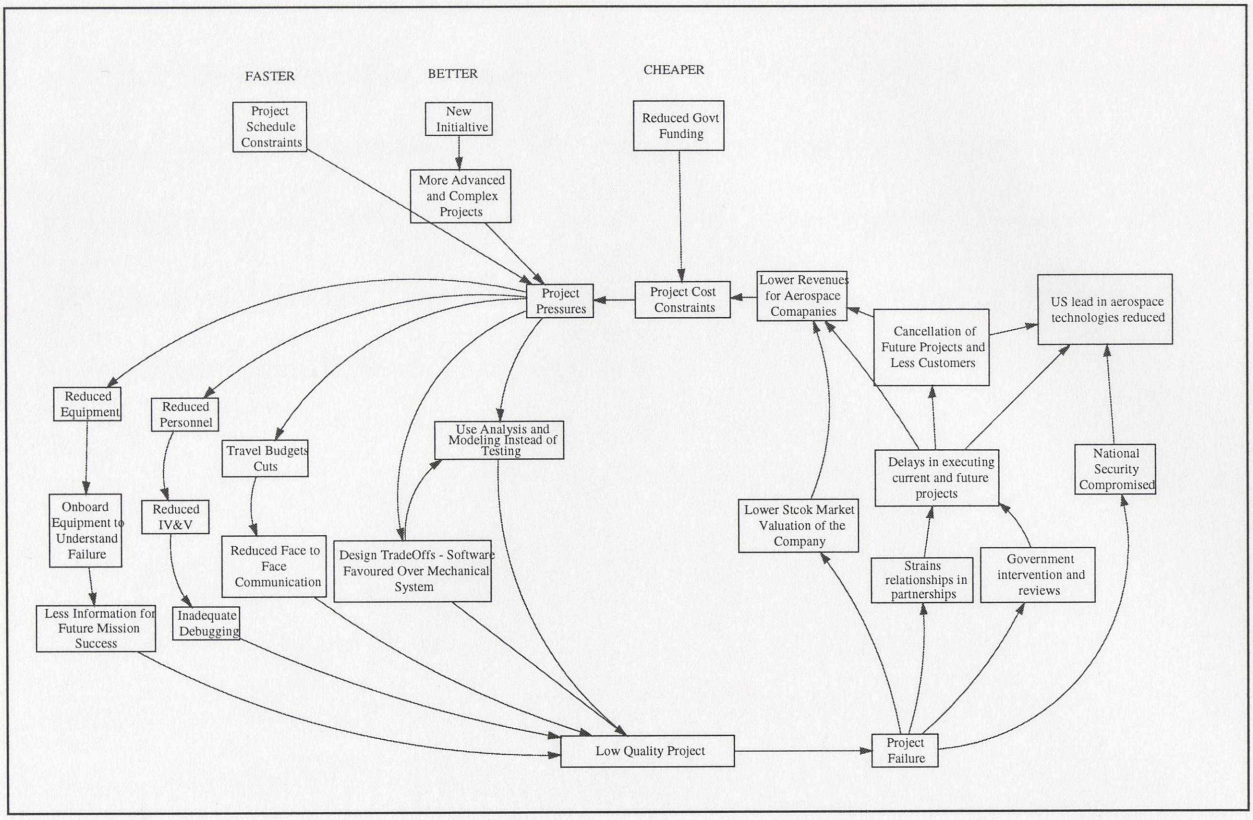


Fig 2.2 Impact of Accidents on Industry

future missions leading to a loss of revenue for the industry. A cycle can be seen with failure leading to lower revenues, which in turn results in lower quality of future projects. The MPL failure reports suggest that reduced personnel and travel budget cuts had lead to communication gaps between project groups. Redundant onboard hardware equipment as well as hardware to understand failures (in-case they occurred) was removed. Thus future missions are endangered due to a lack of thorough understanding of the reasons behind previous failures. The magnitude of the impact of these accidents on society and its potential to hinder progress of humanity requires that a thorough investigation into the actual cause of the accidents be performed. Remedies need to be proposed to check the spread of such accidents in the future. The accidents mentioned previously show that the

quality of software needs to be improved. This cycle of accidents needs to be checked. This goal can be accomplished by either relaxing the faster-better-cheaper mantra or by improving the quality of the projects. One area, which needs attention, is improvement in the quality of software. The next chapter explains why software is different from hardware and proposes the need for better tools.

CHAPTER 3

Root Causes of Accidents

A cursory look at the failures described in the previous chapter may lead to a conclusion that the root cause in most cases is a simple logic error that should have never been included in the design. Even if it were developed during the design process, such errors should have been identified during verification process. Leveson [11] has suggested deficiencies in the safety culture, flawed organizational structures and superficial ineffective technical activities are the root cause of software-related accidents. A careful investigation into aerospace accidents in more recent times shows that the cultural and organizational deficiencies were ignored due to reduced project budgets and tight project schedules. Some very essential safety assessment processes like fault tree analysis were not performed.

3.1 Software is Important

Software is required to interact with a variety of analog and digital components in its environment. Complexity of systems is increasing and software is being used to enable and enhance the capabilities of systems. For example, some advanced systems software includes features like an ability to detect and recover from error conditions in the environment. This is, to a great extent, due to the ease with which functionality can be added. Many hardware behaviors are being implemented in software (for example C130 FCS and MPL thruster control). Due to this unique flexibility offered by software, hardware tests are being replaced with software simulations.

3.2 Software Errors

Software failures have characteristics very different from hardware failures. Classical methods of safety assessment used for mechanical system are not applicable to flight-critical system software. Sufficiently effective means to verify software are not available. Also verification of the simulation is subject to errors in the simulation of the various components. If the functionality were implemented in the hardware, a hardware test would reveal the errors in design. However, if the functionality is implemented in software, it can only be verified in a simulation. Hardware failures (Challenger O ring) can be identified after an accident and preventive measures can be taken after the accident to ensure the same failure does not occur again. Future missions learn from the experience of past failures. However in the case of errors introduced by software, evidence is hard to find and hence future missions are more likely to encounter the same problem since the root cause might either be misdiagnosed or not detected at all. The accident investigations are more thorough and hence expensive and time consuming. Hence missions are delayed, canceled or launched with a higher probability of failure.

Even when the software error is detected, recommendations for containing such errors are limited in scope and not very detailed. Usually the recommendations are stated as "Fix known software problems" or "Fix and validate post-landing fault-recovery algorithm and sequences"[25]. The failure analysis and recommendations treat the errors in software as defects that can be eliminated by standard quality improvement techniques like PDCA (plan do check and act), which is more applicable to hardware manufacturing processes [20]. Quality is assured in the manufacture of hardware components by meeting certain standards in the manufacturing process and a much larger effort is

involved in that process compared to the design phase. Software needs to be developed only once. Hence the highest quality standards need to be applied during the design and design-validation process.

Software errors are specific to the system. Hence remedies suggested by failure reports are specific to the problem. Lessons learned from software-related failures in past projects cannot be easily abstracted. Hence their applicability to new projects and prevention of future failures is not straightforward. This also makes the process of certification of the software difficult because of lack of rational means to predict failure.

Regulations for software meant for reactive control systems like aerospace systems need to be more stringent because these systems have more sophisticated modes of behavior. The software required to control such systems needs to adapt itself to different modes of operation of the system. Trying to design and verify the software through conventional methods like code walk-through and CMM etc. does not help because human comprehension of the number of modes and behaviors of the system is limited and the time available for completing the verification processes is usually inadequate due to tight project schedules to meet market needs (commercial) or launch windows (mars missions).

In addition to the above difficulties, the faster-better-cheaper approach makes it more difficult to develop high quality software.

3.3 Proposed Solution

Aerospace companies have repeatedly stated that their biggest cost drivers for software are requirements errors and the lack of effective requirements analysis

techniques to detect flawed requirements early in development. Existing standards, DO-178 B or IEEE software standards [8], emphasis quality of design processes for validation of aerospace software. They do not address the root cause problem and a solution that is applicable at all phases of the development process. The process of certification is often expensive and time consuming as the current methods of validation are inefficient and ambiguous. The amount of project budget allocated for software also plays an important role in the successful development of software. Nearly one fifth of the total project budget was allotted to the design and development of Boeing 777 software, which has been performing without any serious error since it was deployed. Costs involved in making changes to software grow exponentially over time as the project development proceeds. The following suggestions were made by Leveson [14,15] to improve the quality of software and to integrate safety into the design, validation and verification process.

1. Finding errors early in development so they can be fixed with the lowest cost and impact on the system design.
2. Tracing not only requirements but design rationale throughout the system design and documentation
3. Building required system properties into the design from the beginning rather than emphasizing assessment at the end of the development process when effective response is limited and costly.
4. Supporting the construction of families of related systems and the reuse of the early parts of the system development process.

A system software development tool that is integrated across the entire development cycle of the project and is also applicable at the end-of-life of the project should be considered as a potential alternative to the current methods used in the industry for the development of software.

CHAPTER 4

SpecTRM

Developing system requirements, architectures and translating them to software requirements remains one of the most challenging problems in the design and development of complex reactive control systems. The examples mentioned in chapter 2 and their implications described in chapter 3 show the consequences of failures to meet these challenges. In more recent times the problem is compounded by more complex systems, funding and schedule pressures. Effective design tools have not been available until now to address problems in software requirements specification.

This chapter describes SpecTRM (Specification Tools and Requirements Methodology), which is a CAD system for digital automation [14]. It consists of tools to assist engineers in managing requirements, design and evaluation processes. SpecTRM-RL is a finite-state machine based specification language that represents the evolution of the specification language RSML (Requirements State Machine Language).

The purpose of SpecTRM-RL and the associated tools in the Nimbus environment is to allow analysts to specify safety systems with high reliability. SpecTRM-RL contributes to this goal by being a readable specification language that is usable and understandable by all stakeholders in the specification effort. Therefore, the language is suitable for manual inspections and reviews. Nevertheless, SpecTRM-RL, is a fully formal specification language; thus, analysts can perform formal analysis and simulation

on the requirement model. To achieve a high level of confidence, all three approaches (manual inspections, formal analysis and simulation) must be used in concert. The SpecTRM-RL language and its execution in the Nimbus Environment enable this. Nimbus is one of the most full featured environments available for safety-critical embedded systems development. The simulation and visualization capabilities along with the flexible channel architecture of the environment allows the analyst to connect the Nimbus simulator to many other applications, for example, Microsoft Excel, executable third party software or even actual hardware. To create specifications, the environment includes an editor built on top of Microsoft Word.

Many aerospace projects reuse sub-systems and their design from previous projects in the same program. For example, Ariane 5 has many components that were used in Ariane 4 [21]. Mars Surveyor 2001 Lander will be using parts designed for MCO and MPL [25]. The behavior and interactions of hardware components under varying environmental conditions are better understood and validated through physical tests. Hence, hardware can be reused with a higher degree of confidence than software. Ariane 5 failure has shown that reusing software code is not safe in spite if performing extensive tests. While reuse of code can lead to failures, reuse of specification is a possible alternative. Differences in requirements and constraints between projects can be identified if clear readable specifications are available and an efficient process of converting specifications to code exists.

The Nimbus-tools provide a means to convert specifications directly into executable code. This code can be directly implemented either in simulations or can be used with real hardware. Errors in requirements specifications (incomplete specifications,

contradicting logic, missing conditions etc) can be identified early in the project by performing analysis on the code. Fault-tree analysis is one such method that can aid in identifying fault modes early on in the project and hence mitigate it.

Usually fault-tree analysis is performed after an accident. Information is obtained by the experts from review of the development design, the test programs and interviews with the members of the development and operations team. Sometimes new fault modes are identified during the failure analysis process, which are not taken into account before flight and often result in the accident. In addition to consuming the time and efforts of senior technical experts, who constitute the faultfinding team, the time and efforts of development and operations teams are also wasted. Often the process of scrutiny to determine the fault is not very conducive for building a vibrant and enthusiastic culture that is necessary for a healthy organization.

The above reasons justify the need to perform fault-tree analysis before the launch of a mission. In fact, a fault-tree analysis at regular intervals in the design process is suggested, so that new fault modes identified during any design phase are mitigated by design modifications immediately. If this analysis is performed towards the end the design process, attempts to make design changes will be very expensive and time consuming.

Fig 4.1 shows how the various features of Nimbus are used to implement the SpecTRM methodology.

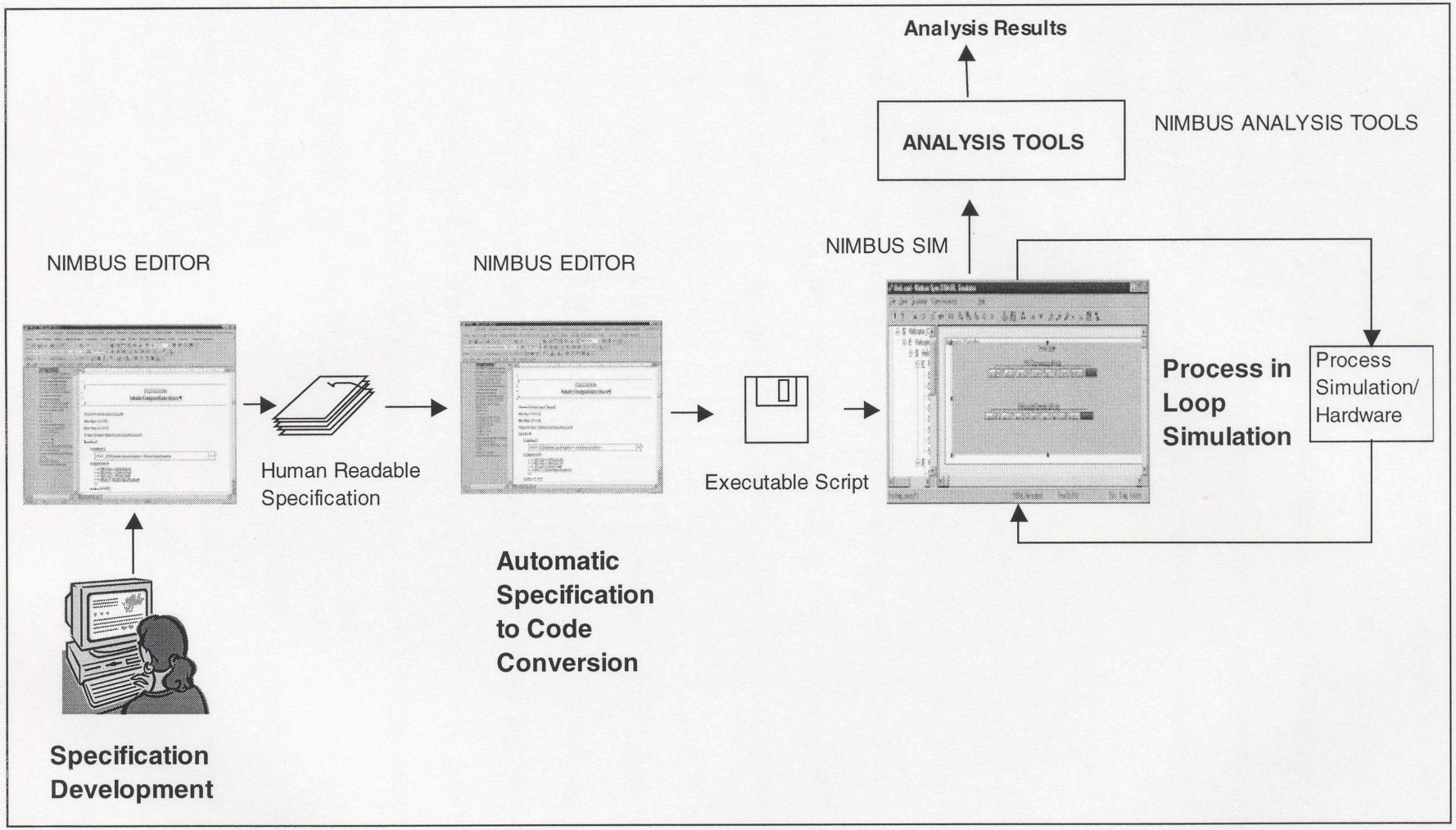


Fig 4.1 Nimbus Environment

The human readable specification provides an elegant and efficient mean of communication for all stakeholders in the project. Efficient and clear communication allows for a clear understanding of the various interactions between the subsystems and with the environment. This allows engineers to identify and eliminate fault modes that can arise from limitations of a sub-system or combinations of subsystems under varying environmental conditions.

Due to the increase in the complexity of systems, limitations on available manpower and time, it is likely that certain fault modes don't get identified. Proposed improvements to the Nimbus-toolset can generate fault-trees and identify fault modes automatically. These tools use the machine-readable code as input. This code can be generated automatically from the human readable specification using the Nimbus-editor. The executable code can be used in a closed loop simulation to identify limitations and errors in the design logic.

By using the above approaches safety gets integrated into the design process. The human readable specifications along with the executable specification and tools provide means to account for the unintended consequences of day-to-day decisions in the design process. This provides a thorough understanding of the system and the software. As a result, confidence in software increases. Engineers would feel more comfortable adding more complex features into the project.

The tools provide the environment to make changes to the system software and understand the implications of these changes with little effort. Limitations in hardware capabilities can be overcome with improved software wherever it is applicable. For example, the software can ignore the false trigger of the touch sensor during the lander-

leg deployment in MPL and hence improvement in the touch sensor hardware design can be avoided.

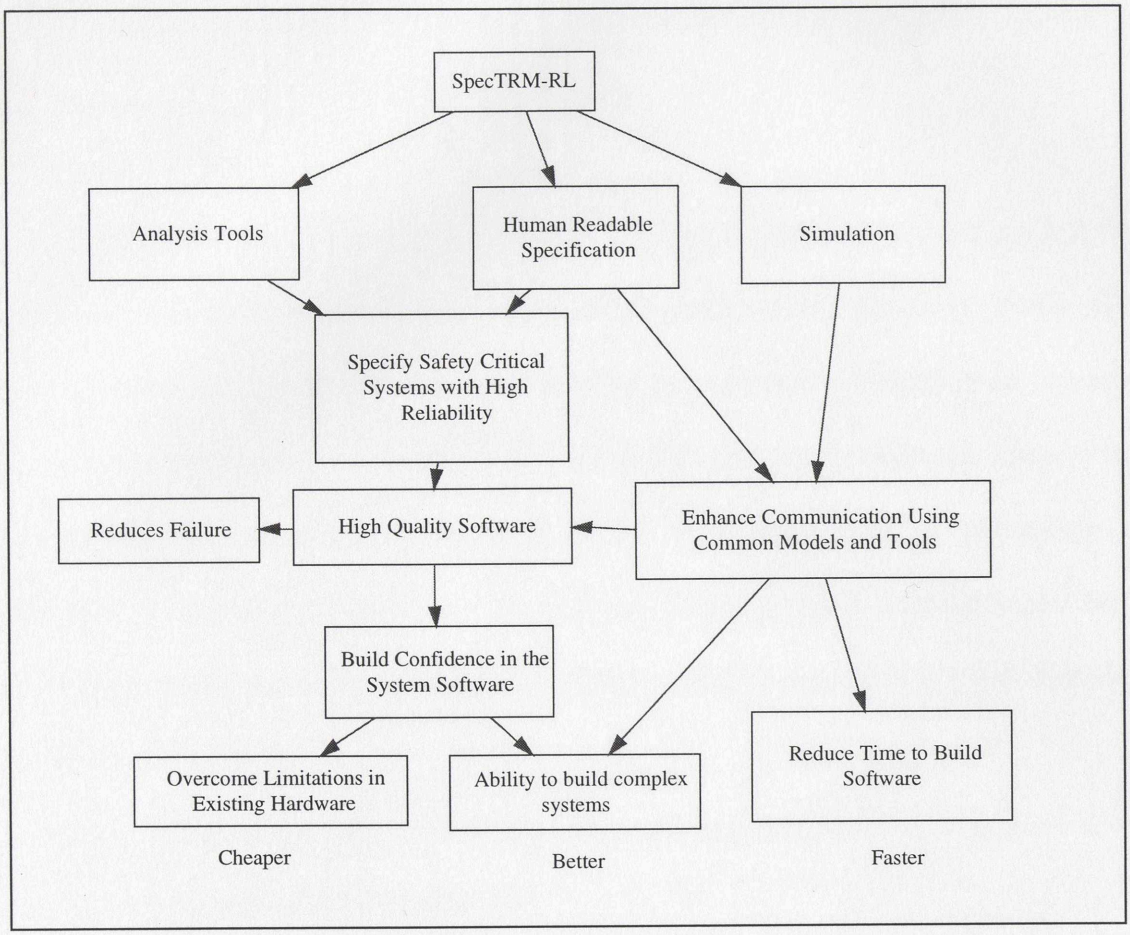


Fig 4.2 Potential Benefits from the Use of SpecTRM

Thus spectrum provides system development platform that integrates across the entire development cycle of the project and is also applicable at the end of life of project. At the heart of the system development process using spectrum is an executable model of the system components. The next chapter describes an example application of Nimbus environment in developing the specification and executable code of an autonomous helicopter system.

CHAPTER 5

Executable Specification of an Autonomous Helicopter System

This chapter describes the development of SpecTRM-RL specifications and the application of SpecTRM methodology to the development of an executable specification of an autonomous helicopter control system. The autonomous helicopter is an example of a complex reactive control system. Many features of this system resemble those of the aerospace systems mentioned in chapter 2. This thesis attempts to take advantage of similarities in the design processes and the features of these systems. Lessons learnt from this example could prove the applicability of this modeling approach to other complex reactive control systems.

The following are the objectives of modeling and analyzing the specifications of the autonomous helicopter control system [15]

- Study the utility of specification modeling language as a design tool to develop safety-critical system.
- Develop a model of the software requirements that could accurately reflect the intended behavior of the autonomous helicopter system
- Use the model to analyze the requirement for design faults in the system.

A spectrum-RL specification consists of a collection of states, transitions, variables, functions macros, constants and interfaces. These constructs are discussed in the following sections with the context of an example: an autonomous helicopter control

system. At a high level the system consists of a flight management system (FMS) on the ground, helicopter on-board guidance controller (HOGC) and a flight control system (FCS). While the FMS and HOGC are discrete controllers, the FCS is a continuous closed loop control system.

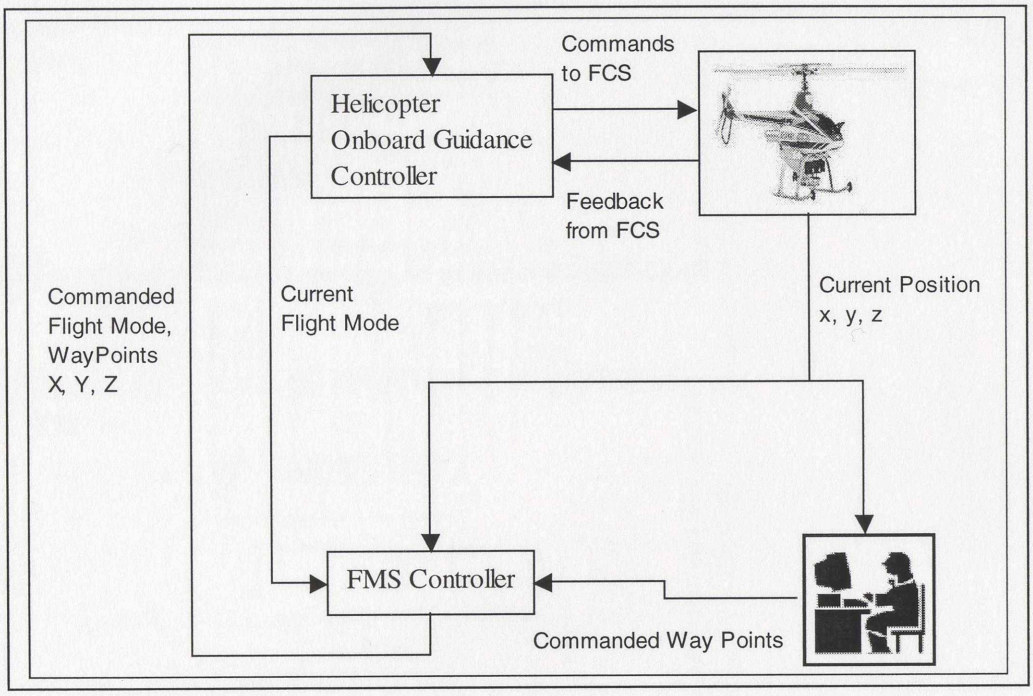


Fig 5.1 A graphical overview of the Autonomous Helicopter System

The objective of the system is to command and monitor the helicopter to travel autonomously to waypoints given by the operator. The human operator can inhibit the autonomous behavior of the control system at any time and manually control the helicopter.

5.1 Autonomous Helicopter System Description

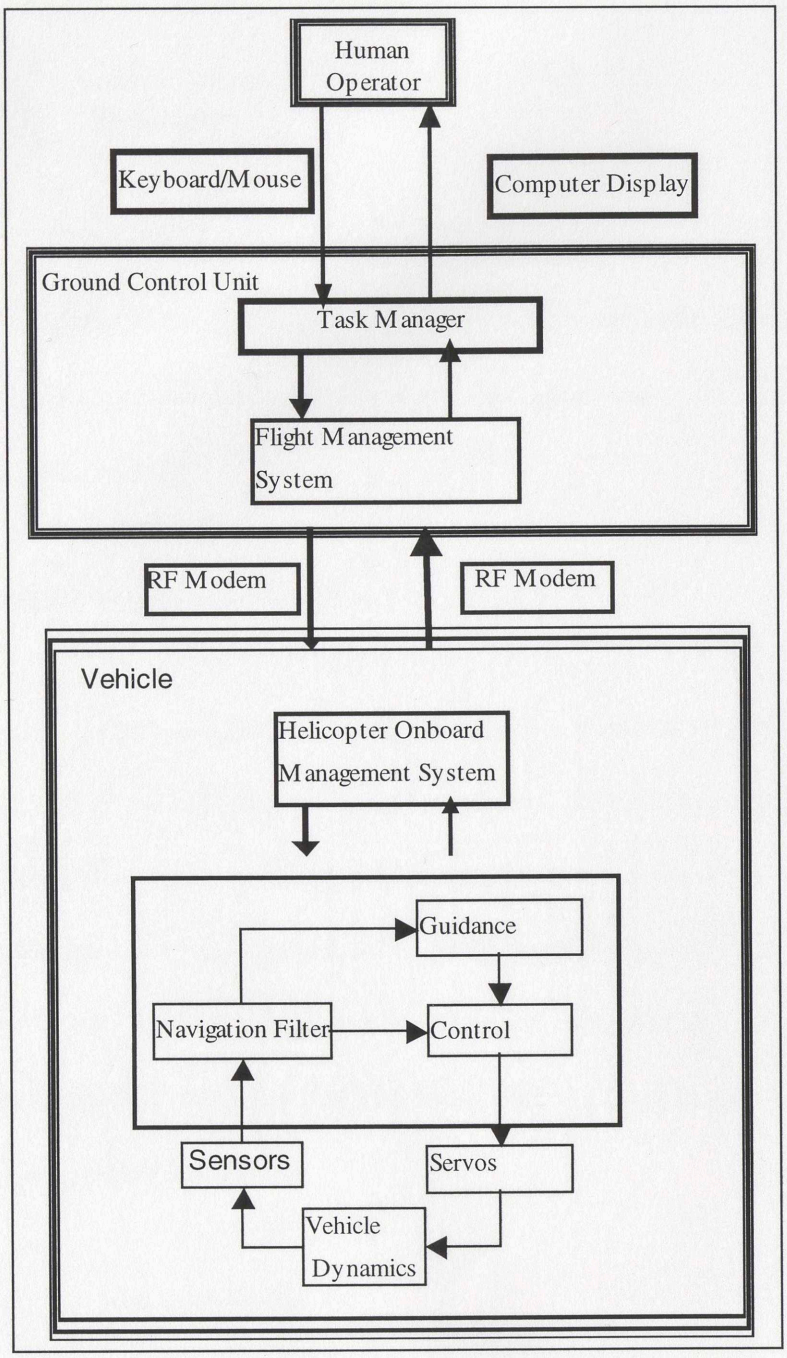


Fig 5.2 Autonomous Helicopter System Architecture

Figure 5.2 is an illustration of the general system architecture of the autonomous helicopter system. It is shown that the system can be broken up into three parts: the helicopter, the Ground Control Unit (GCU), and the operator. The arrows indicate the

flow of information, and the adjacent boxes indicated the type of communications used to transfer information.

5.2 Helicopter

The helicopter houses a variety of on-board hardware systems including a computer processor, various sensors, and a small color camera as payload. The on-board computer runs both the on-board navigation and the flight control software for the helicopter [18].

5.2.1 Navigation System

The navigation system provides an estimate of the helicopter's dynamic state to the FMS through the on-board computer, which takes the navigation solution from the navigation filter and relays that information to the FMS as a vehicle "State Update" message. This information is used by the FMS to determine the next guidance command to send. The navigation filter receives information from an array of sensors, and incorporates them in a Kalman filter, which integrates all the sensor inputs and calculates a navigation solution.

5.2.2 Helicopter Sensors

The sensor elements of the navigation system include a differential global positioning system (DGPS), an inertial measurement unit (IMU), a sonar altimeter, and a compass.

5.2.3 Helicopter Onboard Guidance Controller

The onboard guidance controller translates guidance commands issued from the FMS into commands recognized by the on-board flight control system. The appropriate waypoint commands are sent from the on-board guidance system to the flight control system after guidance commands from the FMS have been processed through the on-board guidance logic. Once the on-board guidance system receives a guidance command from the FMS, it will be responsible for executing that command to ensure that the helicopter reaches the commanded waypoint.

5.2.4 Flight Control System

The control system of the vehicle takes the navigation solution and the translated guidance commands and generates control commands for the vehicle to the servos. There are four independent control loops on the vehicle, which are separated into roll, pitch, yaw and vertical. The control commands control the activity of the six servos on the vehicle, which affect the vehicle's motion.

5.3 Ground Control Unit

The Ground Control Unit (GCU) includes a laptop computer, a joystick and a payload controller. The FMS and GUI software run on the laptop computer. The joystick allows the GCU Operator to directly command vehicle in Pilot Assist Mode. The only communications link from the GCU to the vehicle consists of a radio modem.

It is used for sending FMS commands to the vehicle and receiving vehicle state updates and other on-board information.

5.3.1 Flight Management System

The Flight Management System (FMS) is part of a hierarchical control system responsible for autonomous guidance of the vehicle [19] during mission execution. In addition, it serves as the only interface between the human operator and the helicopter. The FMS sends a variety of commands to the vehicle, but the most important are the guidance commands.

5.4.2 GCU Operator

The GCU Operator has the highest level of supervisory control of the FMS. The primary function of the GCU Operator is to plan an autonomous mission and program the mission into the FMS. Missions are specified by the operator in terms of waypoints.

A secondary function of the GCU operator is to monitor the status of the vehicle based on state-updates, and take the appropriate actions in case of a failure. At any point during mission execution, the GCU operator has the ability to directly command the vehicle via a joystick mounted on the GCU. The operator will be able to command the helicopter's position,altitude and heading while the low-level on-board controller maintains vehicle stability.

5.4.3 Task Manager

The task manager receives the list of waypoint commands from the operator. When the FMS requests the task manager for a waypoint, the task manager provides the next waypoint from the list.

5.5 Information Transmitted

The following information is transmitted to the helicopter by the ground control unit.

5.5.1 Commanded Way Points :

Waypoints are an ordered set of points in space specified by the human operator. The vehicle will execute the mission on straight-line trajectories defined by these waypoints. The waypoints are executed according to their order of entry. A waypoint consists of the following parameters:

- Position in terms of field coordinates (x, y) in ft.
- Altitude above ground level in ft

These waypoints can be entered and executed individually or in the form of a list. The list of waypoints defines the mission to be flown by the FMS during autonomous mission execution.

5.5.2 Commanded Flight Modes

In addition to sending the position, the FMS also sends flight modes. The flight modes are used to determine the overall behavior of the vehicle as it is executing a mission. The flight modes are: Ground Mode, Runup Mode, Takeoff Mode, Waypoint Hover Mode, Waypoint Through Mode, Waypoint Land Mode, and Pilot Assist Mode The flight modes of the vehicle determine the actions of the on-board guidance system, which commands the on-board flight control system.

5.5.2 Supervisory Modes

Piloted, autonomous and return home are the main supervisory modes in which the helicopter can be operating at any time.

5.4 Modeling Method

The modeling of the FMS and the Helicopter Manager is based on an approach, called *intent specification,* developed by Leveson [12]. Intent specifications are hierarchical, and are broken up into five levels, each describing the system in terms of different sets of attributes or language. Following is a brief description of the levels:

- Level 1 (System Purpose) assists system engineers in their reasoning about system-level properties such as goals, constraints, hazards, priorities, and tradeoffs.
- Level 2 (System Principles) allows engineers to reason about the physical principles and laws upon which the design is based.

- Level 3 (Blackbox Behavior) enhances reasoning about the logical design of the system as a whole and the interactions between components as well as the functional state without being distracted by implementation issues.
- Levels 4 (Design Representation) and 5 (Code, Physical Representation) provide the necessary information to reason about individual component design and implementation issues.

The level at which the FMS and the HOGC specifications are written is the third level, also known as the Blackbox Level. At the whole system viewpoint, the blackbox behavior model specifies the system components and their interfaces. A blackbox model of behavior permits statements and observations to be made only in terms of outputs and the inputs that stimulate or trigger those outputs. The model does not include any information about the internal design of the component itself, only its externally visible behavior. Fig 5.3 shows the system-level view of autonomous helicopter, its Ground Control Unit and the Supervisor. The overall system behavior is described by the combined behavior of the components, and the system design is modeled in terms of these component behavior models and the interaction and interfaces between the components [13]. The model is updated and kept consistent with the actual system state through feedback. The HOGC also has a model of its interface with the FMS and the Supervisor. Similarly, the FMS has an internal model of the HOGC and the Supervisor.

A simple assumption about the vehicle's nominal behavior is used throughout the modeling process. The main assumptions are that the vehicle will respond to every request it receives from the FMS, and will execute every guidance command it receives

from the FMS immediately. The vehicle is also assumed to send back a status message at a set time interval of 1 Hz.

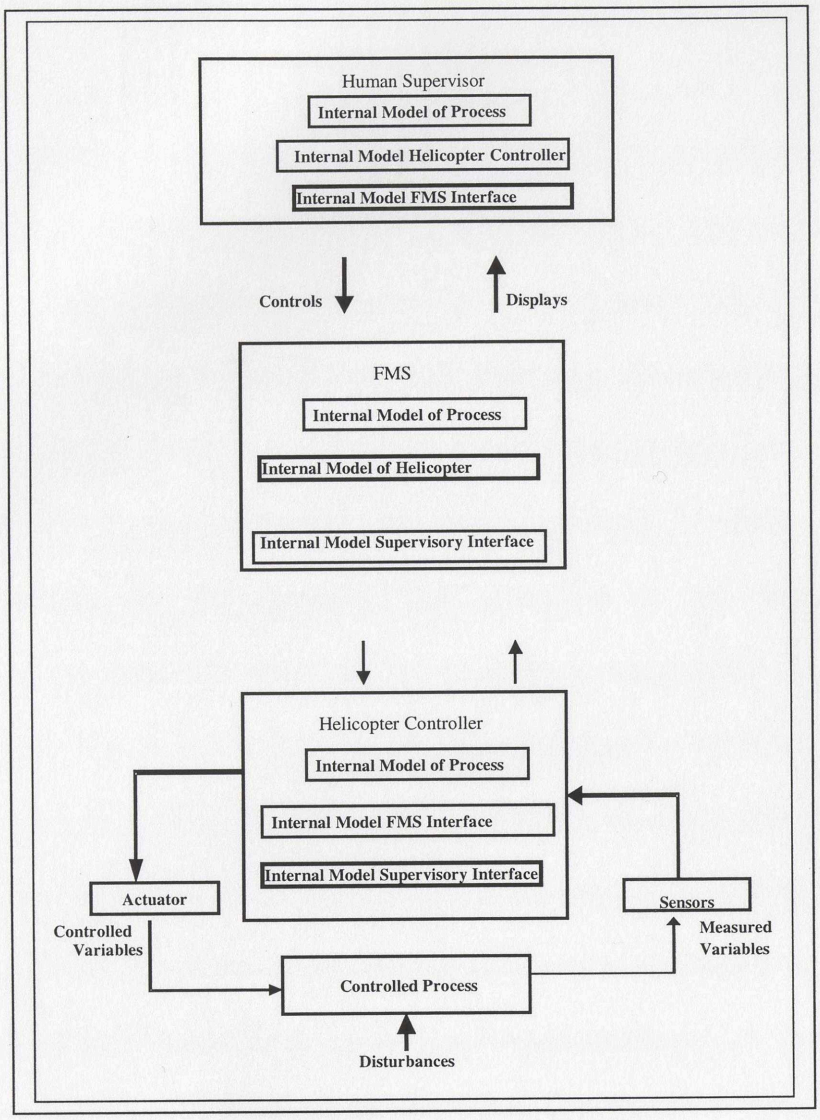


Fig 5.3 Model of the System Specification

5.5 Modeling the Executable Specification

The executable specification for the Draper Autonomous Helicopter was modeled in the Nimbus Environment. Hon Fai Vuong compiled the requirements for the behavior of a part of the system [2]. It provided a reference for understanding the system and

modeling it. Appendix B contains the SpecTRM-RL specifications of the FMS developed with the Nimbus editor. Appendix C contains the machine-readable executable code generated by the Nimbus editor, using the *convert* function, from the human-readable specification.

In autonomous reactive control systems, software is required to interact with a variety of analogue and digital components in its environment. The interfaces between the software and the embedding environment are a major source of costly errors. The FMS specifications include formal definition of interfaces. This separation of interfaces of the specification from the behavior of the specification ensures that no undesirable data enters or leaves the specification (the cause of Ariane 5 failure). The Nimbus environment allows for the execution and evaluation of the formal behavioral specification of the embedded software while it interacts with models of the software's environment [3]. This is achieved by defining the system as a collection of components connected by communication channels. The components are connected to the channels through interfaces and can send messages over the channels. A message is a collection of fields holding the atomic pieces of information communicated between the components. The only information between the components is through unidirectional channels [3].

The executable script for the FMS was loaded into the Nimbus Simulator to form a component that is capable of interacting with models of the task manager, helicopter onboard guidance controller (HOGC) and receives state-updates form the helicopter. The specification was used in a closed loop simulation to verify its behavior.

To illustrate the use of specifications in a closed loop simulation, consider the logic defined in the FMS specification to upload a new waypoint to the helicopter automatically after the helicopter has executed the previous waypoint.

The helicopter flies from one waypoint to the other and receives new waypoint commands and the associated flight mode from the FMS. The FMS keeps tracks the helicopter's current position and determines when a new waypoint needs to be sent to the helicopter. Due to limitations of the onboard sensor accuracies, maneuverability of the helicopter, delays in processing and communications between the helicopter and the ground station, waypoints cannot be executed with pinpoint accuracy. The logic for waypoint guidance is defined such that, if the helicopter reaches an imaginary spherical region around a destination waypoint, execution of that waypoint is considered to be complete. The radius of this imaginary spherical region is defined as the *Radius of Capture*. The part of the SpecTRM–RL specification that defines this logic is as follows. The state variable *EstimatedFlightModeStatus* is set to "Complete" if the function WithInRadOfCapture() returns TRUE, otherwise it is set to "Incomplete".

State Variable

EstimatedFlightModeStatus

Location: FMS_Controller.HelicopterModel.HelicopterState

:= Complete

Condition:

WithInRadOfCapture()	T

:= Incomplete

Condition:

WithInRadOfCapture()	F

The function WithInRadOfCapture is defined as follows.

Macro

WithInRadOfCapture

Parameters: NONE

Condition:

(WPInXVar – LWPInXVar) > - RadOfCapture	T
RadOfCapture > (WPInXVar – LWPInXVar)	T
(WPInYVar – LWPInYVar) > - RadOfCapture	T
RadOfCapture > (WPInYVar – LWPInYVar)	T
(WPInZVar – LWPInZVar) > - RadOfCapture	T
RadOfCapture > (WPInZVar – LWPInZVar)	T

The variables LWPInXVar, LWPInYVar and LWPInZVar are the coordinates of the helicopter obtained in real-time through state updates. WPInXVar, WPInYVar and WPInZVar are the coordinates of the commanded waypoint. RadOfCapture is the *Radius of Capture* which is a constant defined as follows:

Constant

RadOfCapture

Type: INTEGER

Units: ft

Value: 40

When the helicopter enters the imaginary sphere, EstimatedFlightModeStatus (EFMS) transitions from "Incomplete" to "Complete". When the helicopter leaves the imaginary sphere (EFMS) is set to "Incomplete". The transition of EFMS to " Complete" represents the completion of the commanded waypoint. This transition is used by the FMS to request the task manager for the next waypoint. The trigger message (NewWPTriggerMessage) is sent to the Task manager, which then sends the next waypoint to the FMS. The FMS verifies the validity of the waypoint and transmits it to the helicopter. Note that conditions that are checked at the trigger interface before the message is sent require that the EFMS transition from "Incomplete" to "Complete".

Output Interface

NewWPTriggerInterface

Channel: NewWPTriggerChannel

Min Sep: 100 MS

Max Sep: 200 MS

Output Action: SEND(NewWPTriggerMessage)

Handler:

Condition:

..EstimatedFlightModeStatus IN_STATE Complete	T
PREV_STEP(..EstimatedFlightModeStatus IN_STATE Incomplete)	T

Assignments:

1. Trigger := 1

Action: NONE

Fig 5.4 shows the visual output of the closed loop simulation. A visual display of the execution of the logic appears in the Nimbus Simulator. This is in addition to the graphic display for simulated flight of the helicopter. While the helicopter is within the Radius of Capture of the waypoint, the "Complete" state of the EFMS is highlighted. Otherwise the "Incomplete" state is highlighted. This visual display of the execution of the FMS logic along with the simulated flight of the helicopter provides an opportunity to understand the implementation of the logic and its implications to the behavior of the autonomous helicopter.

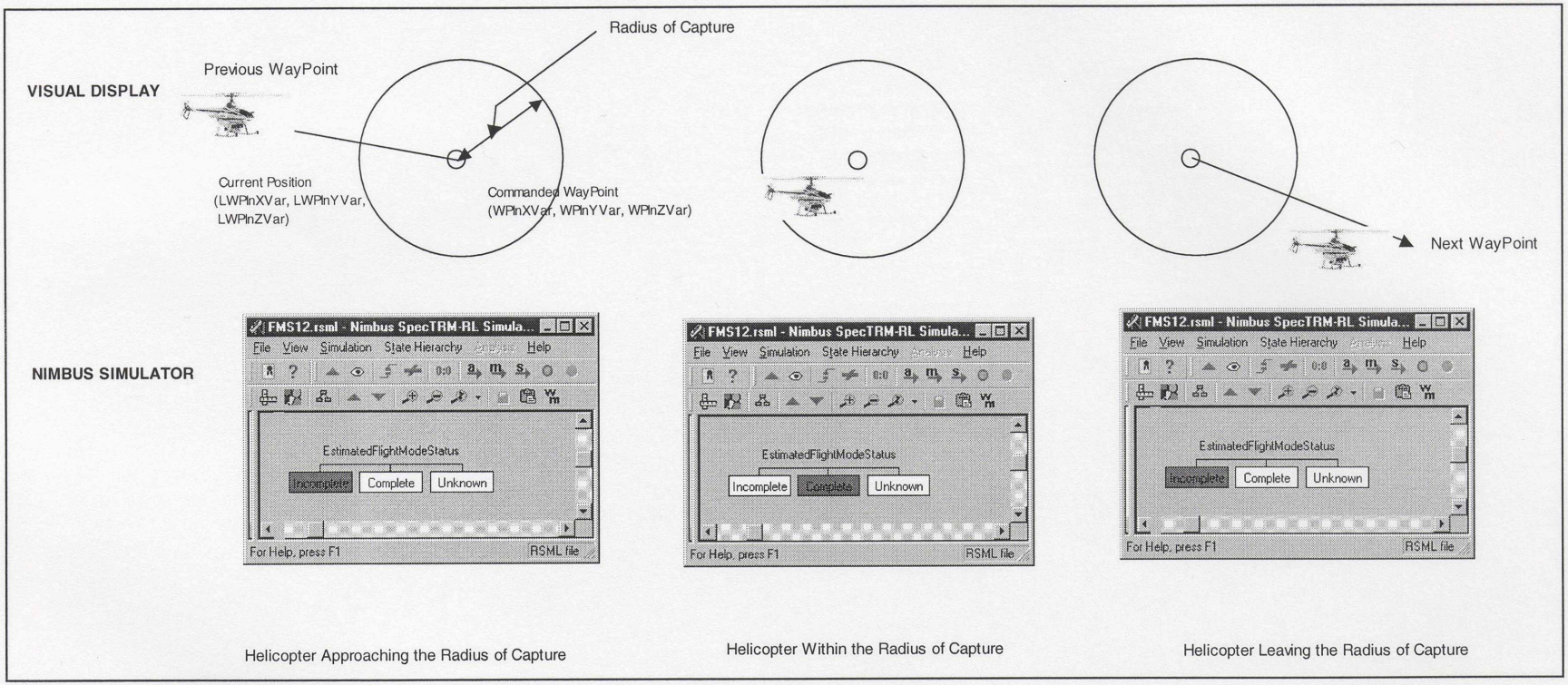


Fig 5.4 Visual display of Logic Implementation in Nimbus Simulator

CHAPTER 6

Simulation Set-Up

This chapter describes the various components that were built and used for a closed loop simulation of the autonomous helicopter system. For a closed loop simulation all the components in the helicopter system should be implemented in hardware or by a software simulation. The FCS simulator developed by Draper Laboratory was used in place of the actual helicopter hardware.

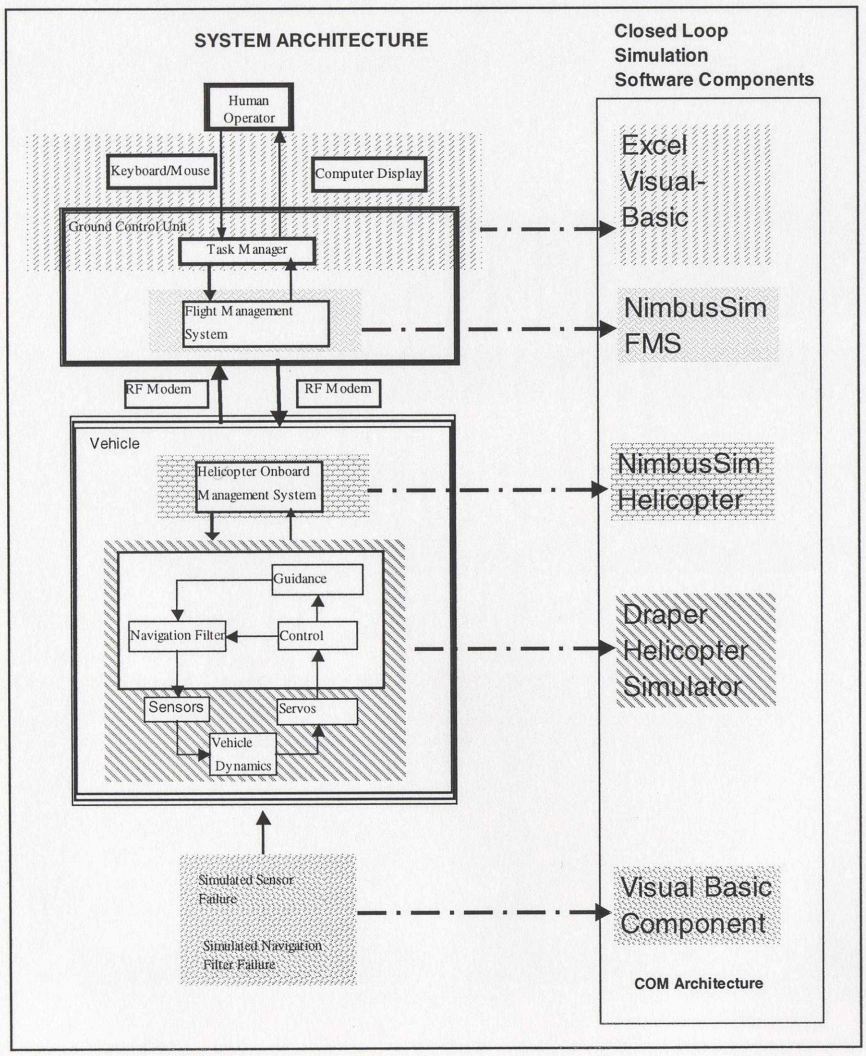
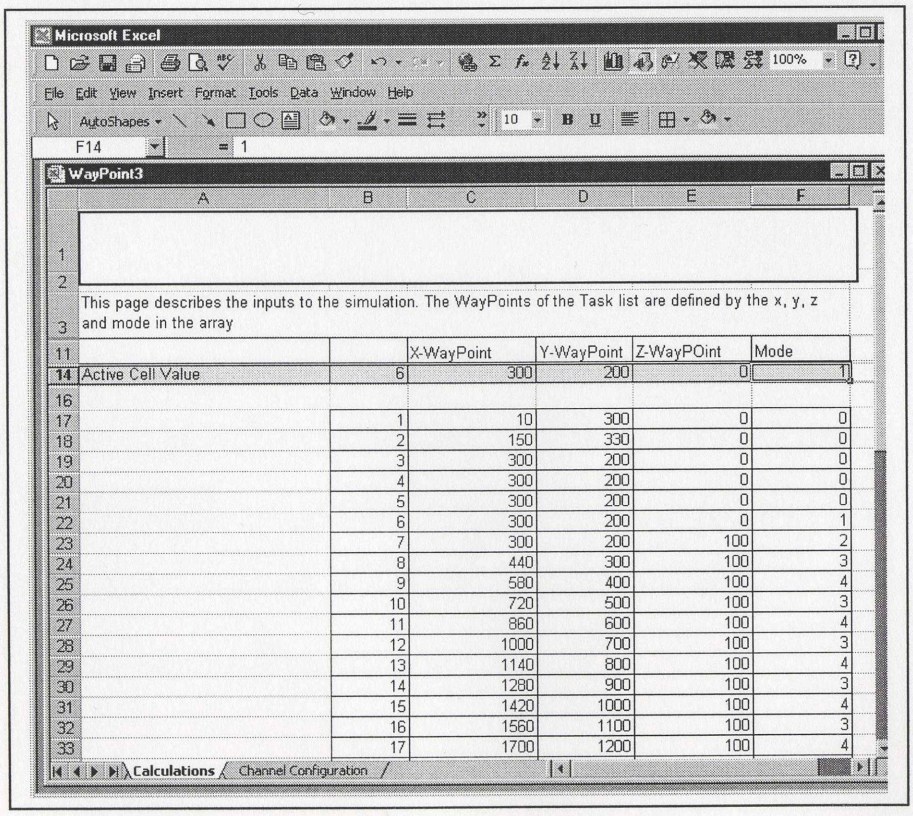


Fig 6.1 Simulation Set-Up

The various elements of the system architecture (shown in Fig 5.2) were implemented in either commercial off-the shelf software packages like Microsoft Excel or as Visual Basic components. Fig 6.1 shows the parts of the system and their corresponding software components used in the closed loop simulation.

6.1 Task Manager Simulation

A Microsoft Excel spreadsheet was used to simulate the functionalities of the Task List. Fig 6.2 shows the list of waypoints (commanded coordinates of the waypoints and the flight mode) that can be manually typed into the spreadsheet. The highlighted cells in the spreadsheet show the current waypoint that is being executed in the simulation.



		X-WayPoint	Y-WayPoint	Z-WayPOint	Mode
Active Cell Value	6	300	200	0	1
	1	10	300	0	0
	2	150	330	0	0
	3	300	200	0	0
	4	300	200	0	0
	5	300	200	0	0
	6	300	200	0	1
	7	300	200	100	2
	8	440	300	100	3
	9	580	400	100	4
	10	720	500	100	3
	11	860	600	100	4
	12	1000	700	100	3
	13	1140	800	100	4
	14	1280	900	100	3
	15	1420	1000	100	4
	16	1560	1100	100	3
	17	1700	1200	100	4

Fig 6.2 Task List Implemented in Microsoft Excel

A Visual Basic (VB) component was developed to simulate the functionalities of the Task Manager. The Task Manager fetches the active waypoint values from the spreadsheet and delivers them to the FMS simulator. The VB component listens on the NewWPTriggerChannel for the trigger message from the FMS. Fig 6.3 shows the VB component as it appears in the simulation.

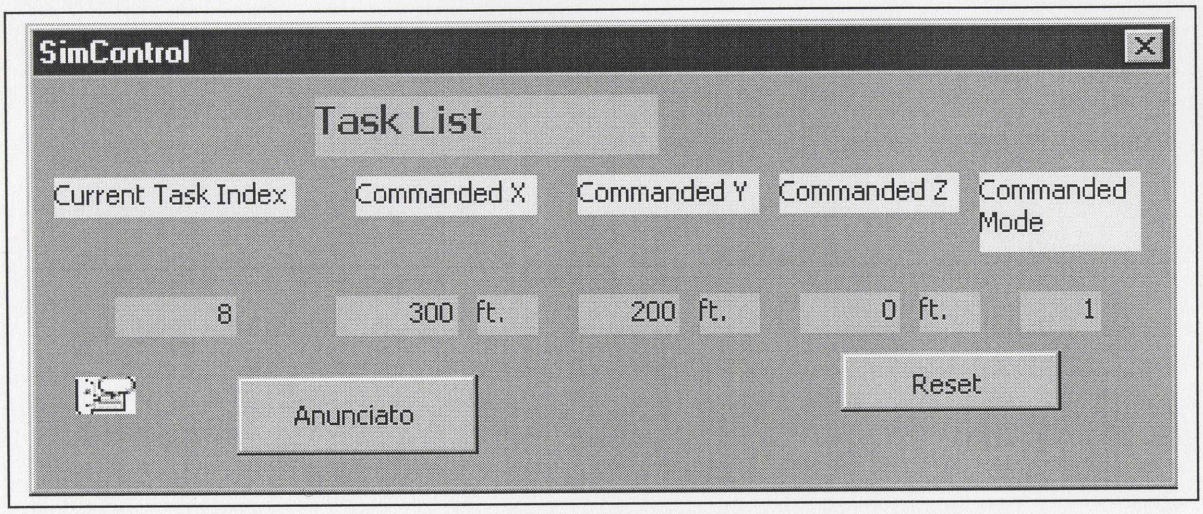


Fig 6.3 Task Manager Implemented as a Visual Basic Component

6.2 Flight Management System (FMS)

The last chapter described the procedure to develop executable specifications. Fig 6.4 shows a screenshot of the FMS specification being executed in the Nimbus Simulator. The executable specification determines when a waypoint execution is complete and requests for waypoint commands from the Task Manager and receives them. It then transmits the commands to the HOGC.

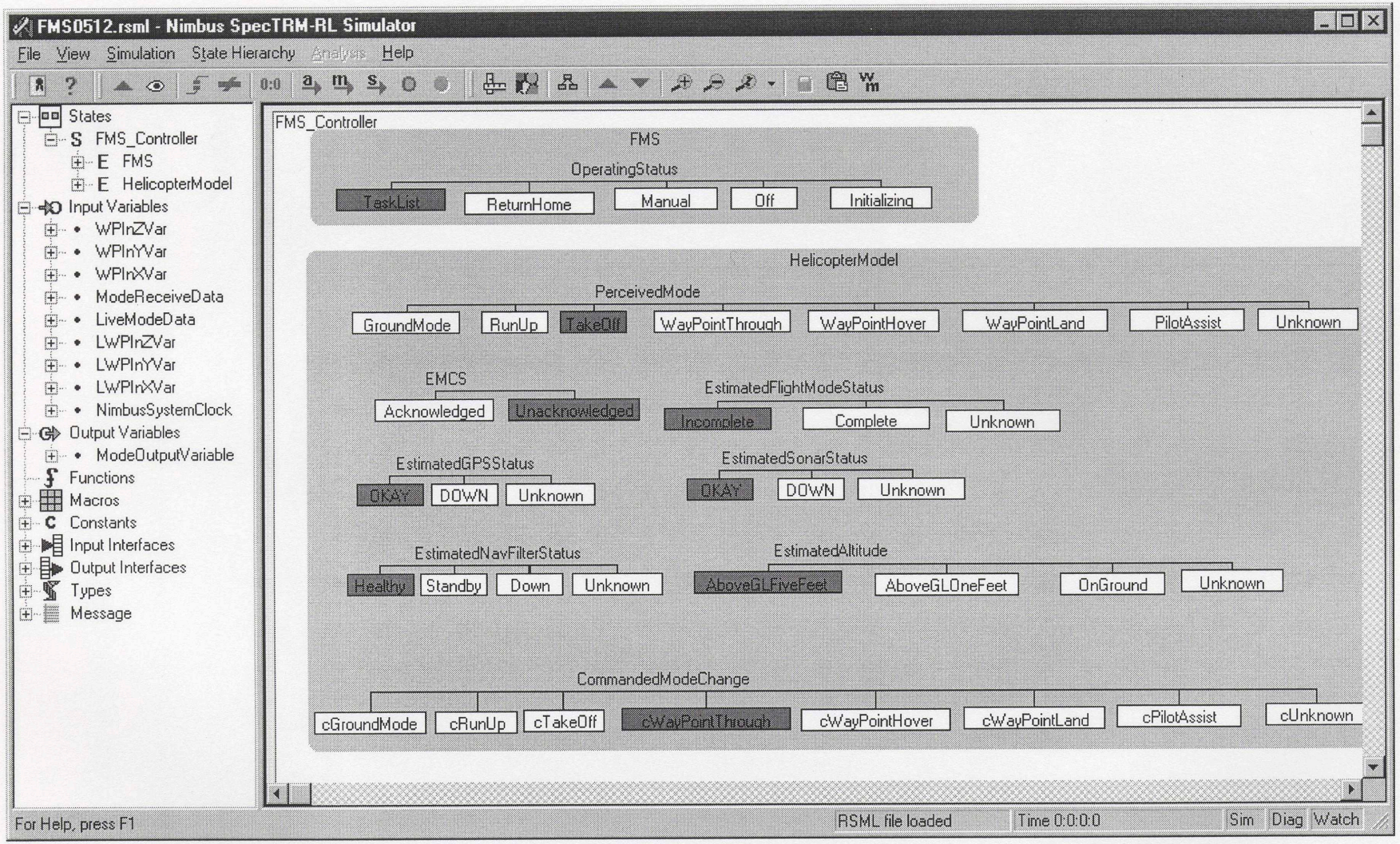


Fig 6.4 FMS Controller in Nimbus Simulator

6.3 Helicopter Onboard Guidance Controller (HOGC)

The implementation of the HOGC component is similar to the FMS. Upon receiving commands, the HOGC delivers them to the FCS.

6.4 Flight Control System (FCS) Simulation

A Draper Helicopter Flight Simulator was used to simulate the FCS. This component takes the waypoint commands and mode as input and returns state-updates as output. Fig 6.5 shows a screenshot of the simulator.

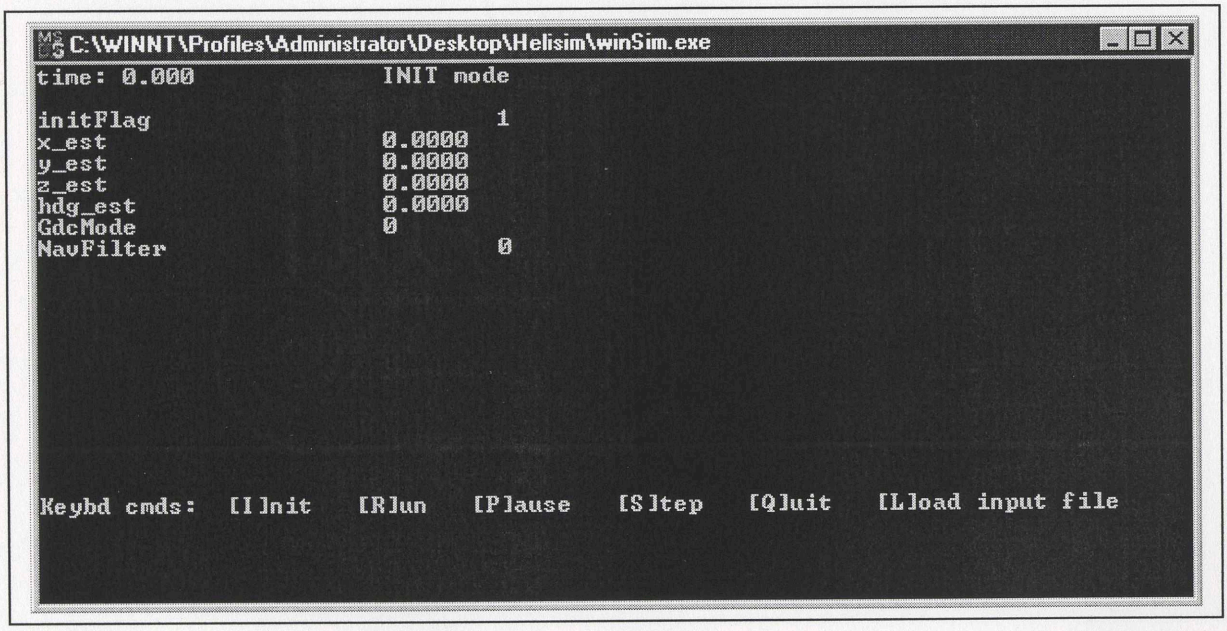


Fig 6.5 Draper Flight Control System Simulator

6.5 Visualization

A WorldUp 3D viewer software was used to display the flight of the helicopter in 3-D. A screen shot of the helicopter in flight during the simulation is shown in Fig 6.6.

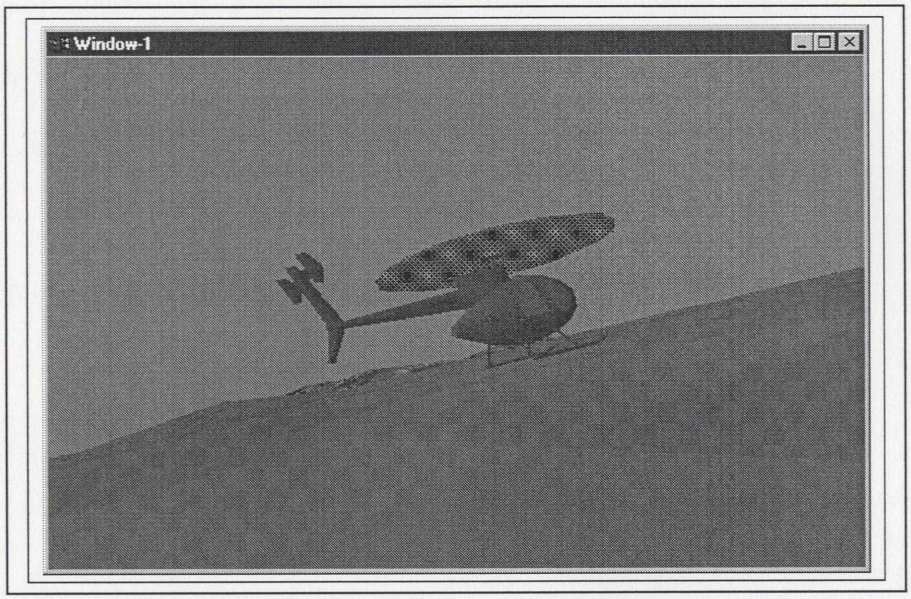


Fig 6.6 WorldUp Visualization of the Helicopter in Flight

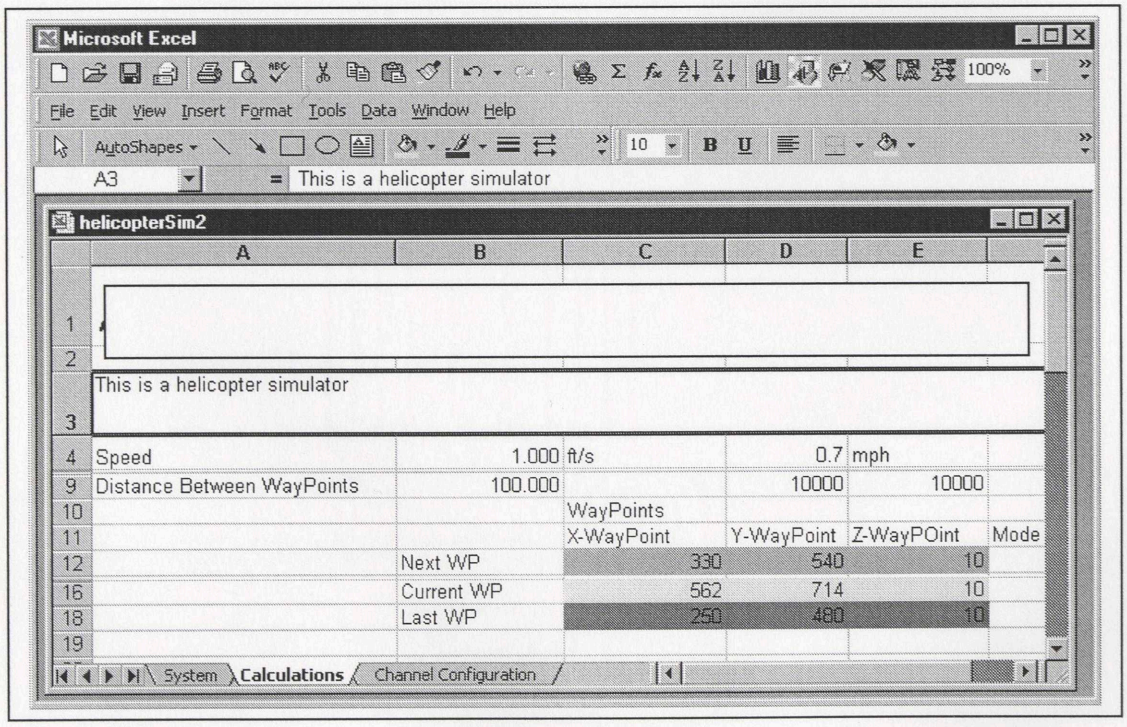


Fig 6.7 Excel based FCS simulation

Until the Draper FCS simulator was available, an excel worksheet was used to implement the functionality of the helicopter FCS. VB was used to program the Excel sheet to receive waypoint commands and return state-updates. Fig 6.7 shows a screen shot of the Excel based simulator. Fig 6.8 shows the VB based 2-D display, which was used to display the flight of the helicopter and the commanded waypoints.

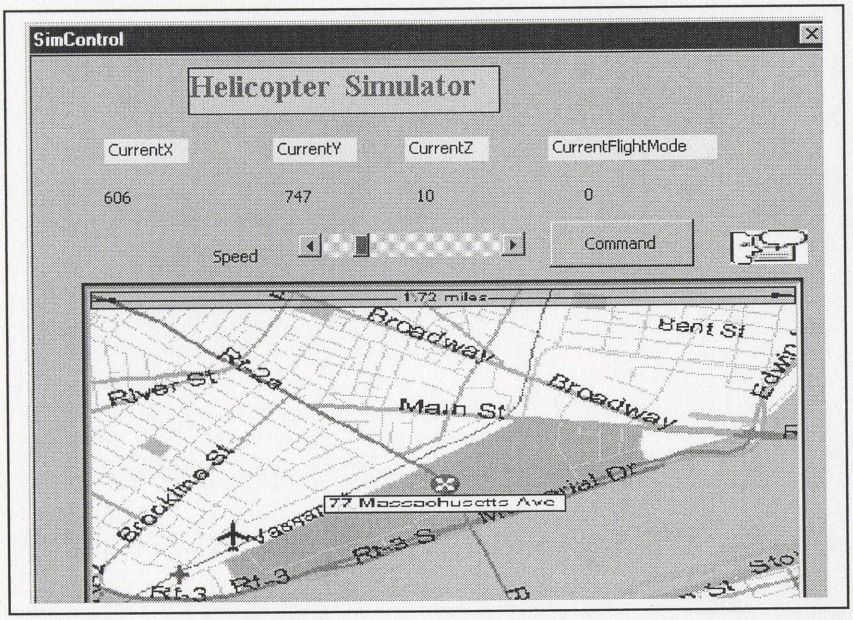


Fig 6.8 Visual Basic Display of the Helicopter and Commanded Waypoints

6.5 Connection Manager

The various components of the simulation could be developed and integrated into a single closed loop simulation due to the ability of the Nimbus Environment to support the Component Object Modeling (COM) architecture. The Nimbus Manager tool was used to connect the components with channels. Fig 6.9 shows a screen shot of the channels and the components connected across them (defined as source and destination) Messages pass through these channels.

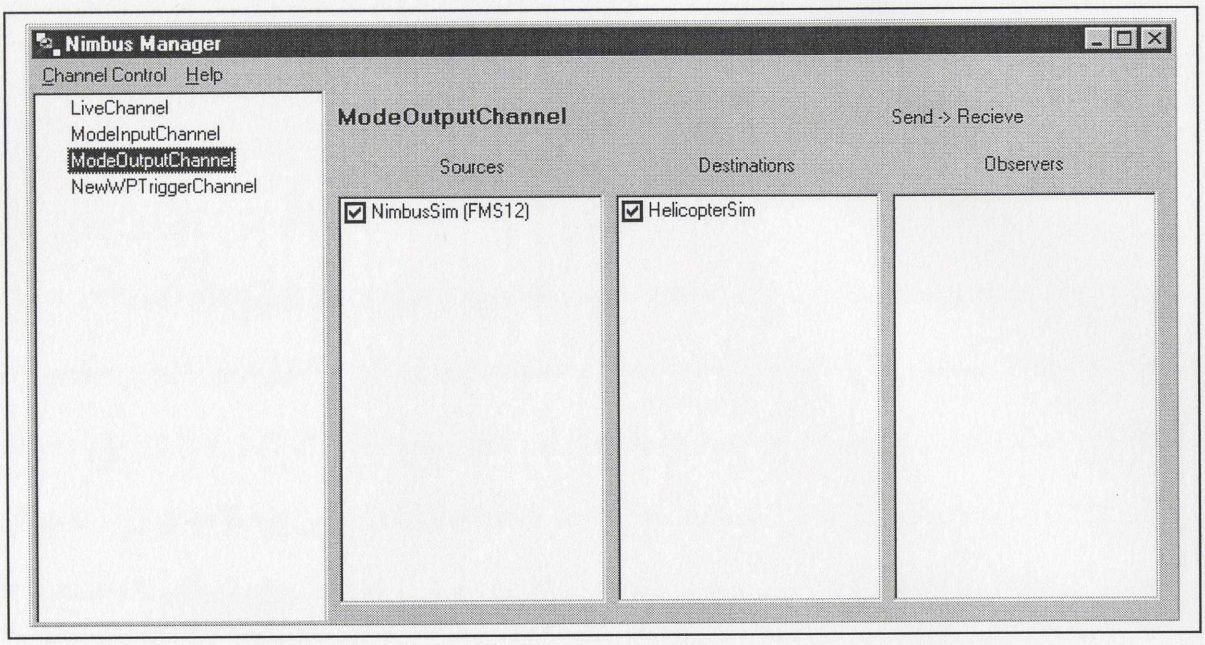


Fig 6.9 Nimbus Channel Manager

6.6 Results

An executable specification of the autonomous helicopter FMS was successfully developed and utilized in a closed-loop simulation using a large number of third party software. The Nimbus environment has proven to be capable of implementing the SpecTRM methodology. However the tool has limited capabilities to perform formal analysis of the specification.

CHAPTER 7

Conclusions

Chapter 2 highlighted the importance of software in aerospace systems. Over the last few years the importance of software has been growing in many other fields involving safety critical systems, such as medical devices, nuclear reactors etc. The unique characteristics of software and software errors described in the context of aerospace systems also plague every safety-critical system controlled by software. Many standards have been proposed to ensure the quality of software and reduce accidents caused by software errors – ISO 9001, CMM, MIL-STD, DOD-STD, IEEE, EIA/IS etc. to name a few. Today systems exhibit sophisticated modes of behavior, which are unique to the system and its application. Software is used to implement these numerous modes of behavior. Hence software developed for one system is different from that of the others. So are the errors in software. Hence applying any one standard to qualify software has not been successful. Ambiguities and problems are often encountered in adhering to standards, as the standards themselves have not been sufficiently abstracted to be applicable to a wide range of system software. FAA and other regulatory agencies further exacerbate the ambiguities due to a lack of harmony in interpreting the standards.

Clarifications to the proposed guidelines, intent and application of these standards are often sought. Due to the fast changing nature of software technology, standards are finding it difficult to keep up with the developments in the industry. These problems were realized and guidelines were imposed on software life-cycle processes to ensure that software development results in safe system. CMM and ISO 9001 emphasize

qualification of software development processes. Management responsibility, audits training etc. have been identified. RTCA/DO-178B provides guidelines for the production of airborne system equipment software. It is used internationally to specify the safety and airworthiness of software for avionics systems [23]. It describes the integrity and readability of such software. However in spite of adhering to these standards, accidents have been witnessed.

Most of these accidents are due to requirements errors, not coding errors. Documentation throughout the design process has been emphasized to maintain traceability of requirements. Software code is difficult to read. Hence, in spite of proper documentation, tracing requirements to implementation in code is challenging. This approach is necessary but not sufficient to establish evidence for the absence of critical design errors.

Chapter 3 described the implications of aerospace related accidents. Software related accidents have a greater impact than hardware failures since software is often perceived to be easy to develop. Unlike hardware failures, the general populace considers errors in software avoidable. This tarnishes the image of NASA and other aerospace industries, which have traditionally been considered to have the best brains in the country. Such a loss of face can significantly reduce the popularity ratings leading to a reduced support in the congress for NASA [16]. This would mean reduced funding for the agency for future missions. Investors shy away from aerospace companies due to similar reasons. Therefore, alternate means to qualify safety critical software needs to be identified.

The previous chapters have shown the applicability of the SpecTRM methodology to the design of specifications for an autonomous helicopter. This thesis proposes the SpecTRM methodology as a means to ensure development of safe software. It integrates safe design practices across the entire design cycle. The SpeTRM-RL specification provides clear documentation that is human readable and hence can be used to verify the requirements. Automated code generation ensures that the code generated traces to the requirements in the specification accurately. The formal analysis tools prove the completeness and accuracy of requirements and hence validate the design. Simulation of the executable specification provides an effective means to communicate and verify the implementation of the logic.

One of the most interesting features of SpecTRM is the ability to make changes easily in the specification and verify the implementation of the logic immediately. For example the value of the *Radius of Capture* (mentioned in chapter 4) can be changed or an additional condition to trigger the transition from " Incomplete" to "Complete" can be made in the word document (human readable specification). Using the Nimbus Editor tool this modified document can be converted into an executable script, loaded into the Nimbus Simulator and executed in a closed loop simulation that was already set up for the previous version of the specification. The simulation would show the changes made in the specification logic. Once all components of the simulation set-up are built and integrated, designers can try various means of implementing the logic to obtain the desired behavior of the system. Safer designs can be identified and hence improve the quality of software and probability of success of the mission.

Chapter 3 mentioned some of the recommendations of failure reports such as "Fix known software errors". That raises the question " What about logic errors that are unknown?" Formal analysis tools can be used to identify the condition in which the system can get into a hazardous state. The logic can be refined until all the hazardous conditions are eliminated and all stakeholders feel comfortable with the changes. The changes reflect in the human-readable specification. Thus documentation is constantly kept updated. The simulation verifies the intended behavior of the system and the analysis tools provide a formal proof of the safety of software.

All the above advantages can be obtained within a short time. Hence these tools can be implemented throughout the design process. Safety features can be incorporated into design as the design evolves. The tools also allow for the reusability of the specification by making modifications to reflect the changes in requirements specifications developed with SpecTRM-RL. The simulation set-up can also be reused with minor modifications to reflect the changes in the environment.

Readable specification with an ability to automatically convert into executable code and implementation in hardware has many other advantages. Large-scale software development involves a large number of programmers who need to understand how the part of the code developed by them affects the rest of the system software. With a clear definition of interfaces, components, communication channels and messages, software development effort can be distributed among groups. Integration of the various parts of the specification is enabled by clear definition of interfaces of the components.

Suggested future work:

A system dynamics model is suggested that would include the various factors leading to low quality software and the consequences of failures as mentioned in Fig 2.2. Other factors, such as defense projects and foreign aerospace projects should also be considered in the model as these programs suffer from the same problems mentioned in Chapter 2 and Chapter 3. The short term and long term benefits from the use of SpecTRM methodology can then be quantified from the results of the system dynamics model. Intangible loses incurred due to accidents such as decline in innovation, stigma associated with failure and its impact on work culture, tensions within partnerships, project delays etc. should be quantified through research and interviews for the system dynamics model. Due to the feed forward nature of the accidents as described in Chapter 2, the system dynamics model will help identify means to check future accidents. Alternate means can be identified, such as reducing project pressures through increased budgets and more relaxed project schedules or a need for collaboration on development of safe practices between the various players in the aerospace community (commercial, foreign and defense). These means are usually constrained by other factors, such as competition, defense and corporate secrets etc. and hence cannot be implemented with ease. The proposed methodology suggested in this thesis can achieve the above objective.

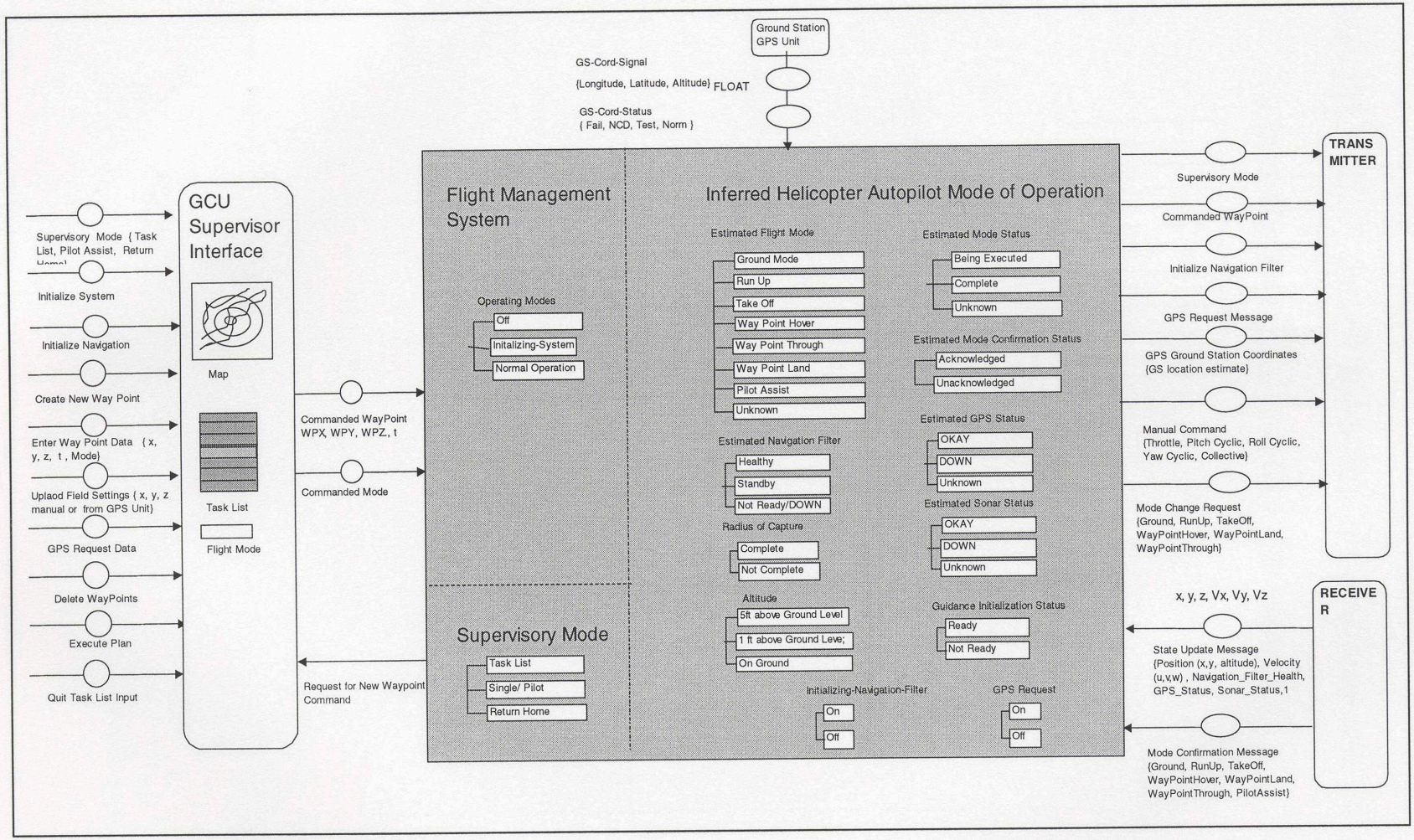
The Nimbus Environment has a potential to drastically reduce accidents in future complex reactive-control systems through the development of safe software. However the toolset in the current Nimbus environment is incomplete. Efforts and resources need to be devoted towards expanding the current capabilities and improving upon them.

REFERENCES

1. G Auvray , *Methods anbd Facilities for Qualifying the Electrical Systems and Software of the Ariane 5 European Launch Vehicle*, AIAA 93-4544, Computing in Aerospace Conference, Oct 19-21, 1993, San Diego, CA.
2. Hon Fai Vuong, *Modeling and Analysis of Software Specifications for an Autonomous Aerial Vehicle*, Masters Thesis, Department of Aeronautics and Astronautics, MIT, May 1999.
3. Jeffery Michael Thompson, *Nimbus: A framework for static analysis and simulation of system-level intercomponent communication*, Master's thesis, Dec 1999, University of Minnesota.
4. Jeffrey M. Thompson , *SpecTRM-RL Language Manual*, Safeware Engineering Corporation,Seattle, Washington and the Critical systems Research Group, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, Minnesota.
5. Jeffrey M. Thompson, *NimbusChannel Users Guide,* Safeware Engineering Corporation,Seattle, Washington and the Critical systems Research Group, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, Minnesota.
6. Jim Schefter, *NASA's Changing Fortunes* <http://www.popsci.com/scitech/features/nasa_fortunes/index.html>
7. Kimberly J. Williamson, *NimbusEdit Manual*, Safeware Engineering Corporation,Seattle, Washington and the Critical systems Research Group, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, Minnesota.
8. Leslie A (Schad) Johnson, *DO- 178 B, "Software Considerations in Airborne Systems and Equipment Certification,"* Flight Systems, Boeing Commercial Airplane Group.
9. Lions, J.L. Ariane 5 Flight 501 Failure Report, Paris, July 1996 <http://www.esa.int/htdocs/tidc/Press/Press96/ariane5rep.html>
10. Mark C. Paulk, *How ISO 9001 compares with the CMM*, IEEE Jan 1995, 0740-7459.
11. Nancy G. Leveson, *Safeware: System safety and Computers*, Addison-Wesley publishing company, Sep 1995.
12. Nancy G. Leveson , *Intent Specification: An Approach to Building Human-Centered Specifications.*
13. Nancy G. Leveson, M. Heimdahl,H. Hildreth, J. Reese, R. Ortega, *TCAS II System Requirements Specification*, Draft, December 1992.
14. Nancy G. Leveson, Jon Damon Reese, Mats P.E. Heimdahl *SpecTRM: A CAD System for Digital Automation*

15. Nancy G. Leveson, *Requirements Analysis for Safety*, Project Proposal to Draper Laboratories, 1999.
16. Robert Dare, Melinda Delaney, Anand Karasi, Vikas Mehta, Jeffrey Munson, Matt Nuffort, Ryan Schaefer, Larry Siegel, Annalisa Weigel, Julie Wilhelmi, Brandon Wood, *From the Moon to Mars : Apollo-SEI Project*, Space Policy Term Project Report, 16.899a, May 2000
17. Robert H. Sfarzo, *NimbusSim Graphical User Interface Manual*, Safeware Engineering Corporation, Seattle, Washington and the Critical systems Research Group, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, Minnesota.
18. Sanders, Christopher P., *Real-time Collision Avoidance for Autonomous Air Vehicles*. SM thesis, Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, Cambridge, MA, January 1998.
19. Sanders, Christopher P., DiBitteto, Feron, E., Vuong, H., Leveson, N. *A Hierarchical Control of Small Autonomous Helicopters* 37th IEEE Conference on Decision and Control, Tampa, FL, December 1998.
20. Shoji Shiba, Alan Graham, David Walden, *A New American TQM - Four Practicle Revolutions in Management*, Productivity Press, Portland, Oregon.
21. *Ariane 5 Report Details Software Design Errors* , Aviation Week and Space Technology, Sep 9 , 1996, page 79
22. *Department of Defense Assessment of Space Launch Failures: Executive Summary*<http://www.af.mil/lib/misc/spacebar99b.htm>
23. *Notes on DOE 178 B: Software Considerations in Airborne Systems and Equipment Certification* <http://www.cs.cmu.edu/~koopman/depend/book/rtca92.htm>
24. *The Nimbus Environment for Specification of Safety Critical Systems*, Safeware Engineering Corporation,Seattle, Washington and the Critical
25. *Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions*, JPL Special Review Board Nasa Mars Failures:
26. *Roger Comminsion Report* ,<http://www.panix.com/~kingdon/space/rogers-c.html>

APPENDIX A - Flight Management System SpecTRM Internal Model



APPENDIX B – SpecTRM-RL Specification of Autonomous Helicopter System

---

Output Interface

NewWPTriggerInterface

Channel: NewWPTriggerChannel

Min Sep: 100 MS

Max Sep: 200 MS

Output Action: SEND(NewWPTriggerMessage)

Handler:

Condition:

..EstimatedFlightModeStatus IN_STATE Complete	T
PREV_STEP(..EstimatedFlightModeStatus IN_STATE Incomplete)	T

Assignments:

Trigger := 1

Action: NONE

Message

NewWPTriggerMessage

Fields:

Trigger is INTEGER

Output Interface

ModeOutputInterface

Channel: ModeOutputChannel

Min Sep: 100 MS

Max Sep: 200 MS

Output Action: SEND(ModeOutputMessage)

Handler:

Condition:

PREV_STEP(ModeOutputVariable) != ModeOutputVariable	T

Assignments:

WPOutX := WPInXVar
WPOutY := WPInYVar
WPOutZ := WPInZVar
MOD := ModeOutputVariable

Action: NONE

Message

ModeOutputMessage

Fields:

WPOutX is INTEGER
WPOutY is INTEGER
WPOutZ is INTEGER
MOD is ModeType

Output Variable

ModeOutputVariable

Type: ModeType

:= ModeType::kGroundMode **IF**

..CommandedModeChange IN_STATE cGroundMode	T

:= ModeType::kRunUp **IF**

..CommandedModeChange IN_STATE cRunUp	T

:= ModeType::kTakeOff **IF**

..CommandedModeChange IN_STATE cTakeOff	T

:= ModeType::kWayPointThrough **IF**

..CommandedModeChange IN_STATE cWayPointThrough	T

:= ModeType::kWayPointHover **IF**

..CommandedModeChange IN_STATE cWayPointHover	T

:= ModeType::kWayPointLand **IF**

..CommandedModeChange IN_STATE cWayPointLand	T

:= ModeType::kPilotAssist **IF**

..CommandedModeChange IN_STATE cPilotAssist	T

:= ModeType::kUnknown **IF**

..CommandedModeChange IN_STATE cUnknown	T

Input Interface

InitializeSystemInterface

Channel: InitializeChannel

Min Sep: 50 MS

Max Sep: 100 MS

Input Action: RECEIVE(InitializeMessage)

Receive_Handler:

Condition: TRUE

Message

InitializeMessage

Fields: NONE

State Variable

OperatingStatus

Location: FMS_Controller.FMS.FMS_Dummy

Transition: Off → Initializing

Condition:

MESSAGE_AT(InitializeSystemInterface)	T

Message

ModeReceiveMessage

Fields:

WPInX is INTEGER
WPInY is INTEGER
WPInZ is INTEGER
MRD is ModeType

Input Interface

ModeReceiveInterface

Channel: ModeInputChannel

Min Sep: 50 MS

Max Sep: 100 MS

Input Action: RECEIVE(ModeReceiveMessage)

Receive_Handler:

Condition: TRUE

Assignments:

WPInXVar := WPInX
WPInYVar := WPInY

WPInZVar := WPInZ
ModeReceiveData := MRD

Input Variable

ModeReceiveData

Type: ModeType

Initial Value: Undefined

Input Variable

WPInXVar

Type: INTEGER

Units: meters

Initial Value: Undefined

Expected Min: 0

Expected Max: 10000

Input Variable

WPInYVar

Type: INTEGER

Units: meters

Initial Value: Undefined

Expected Min: 0

Expected Max: 10000

Input Variable

WPInZVar

Type: INTEGER

Units: meters

Initial Value: Undefined

Expected Min: 0

Expected Max: 10000

Message

LiveStatusMessage

Fields:

LWPInX is INTEGER
LWPInY is INTEGER
LWPInZ is INTEGER
LMRD is ModeType

Input Interface

LiveStatusInterface

Channel: LiveChannel

Min Sep: 50 MS

Max Sep: 100 MS

Input Action: RECEIVE(LiveStatusMessage)

Receive_Handler:

Condition: TRUE

Assignments:

LWPInXVar := LWPInX
LWPInYVar := LWPInY
LWPInZVar := LWPInZ
LiveModeData := LMRD

Input Variable

LiveModeData

Type: ModeType

Initial Value: Undefined

Input Variable

LWPInXVar

Type: INTEGER

Units: meters

Initial Value: Undefined

Expected Min: 0

Expected Max: 10000

Input Variable

LWPInYVar

Type: INTEGER

Units: meters

Initial Value: Undefined

Expected Min: 0

Expected Max: 10000

Input Variable

LWPInZVar

Type: INTEGER

Units: meters

Initial Value: Undefined

Expected Min: 0

Expected Max: 10000

State Variable

EstimatedFlightModeStatus

Location: FMS_Controller.HelicopterModel.HelicopterState

:= Complete

Condition:

WithInRadOfCapture()	T

:= Incomplete

Condition:

WithInRadOfCapture()	F

State Variable

PerceivedMode

Location: FMS_Controller.HelicopterModel.HelicopterState

:= GroundMode

Condition:

Message_At (LiveStatusInterface)	T
LiveModeData = ModeType::kGroundMode	T

:= RunUp

Condition:

Message_At (LiveStatusInterface)	T
LiveModeData = ModeType::kRunUp	T

:= TakeOff

Condition:

Message_At (LiveStatusInterface)	T
LiveModeData = ModeType:: kTakeOff	T

:= WayPointHover

Condition:

Message_At (LiveStatusInterface)	T
LiveModeData = ModeType::kWayPointHover	T

:= WayPointThrough

Condition:

Message_At (LiveStatusInterface)	T
LiveModeData = ModeType::kWayPointThrough	T

:= WayPointLand

Condition:

Message_At (LiveStatusInterface)	T
LiveModeData = ModeType::kWayPointLand	T

:= PilotAssist

Condition:

Message_At (LiveStatusInterface)	T
LiveModeData = ModeType::kPilotAssist	T

State Variable

CommandedModeChange

Location: FMS_Controller.HelicopterModel.HelicopterState

:= cGroundMode

Condition:

Message_At (ModeReceiveInterface)	T
ModeReceiveData = ModeType::kGroundMode	T

:= cRunUp

Condition:

Message_At (ModeReceiveInterface)	T
ModeReceiveData = ModeType::kRunUp	T

:= cTakeOff

Condition:

Message_At (ModeReceiveInterface)	T
ModeReceiveData = ModeType:: kTakeOff	T

:= cWayPointHover

Condition:

Message_At (ModeReceiveInterface)	T

ModeReceiveData = ModeType::kWayPointHover	T

:= cWayPointThrough

Condition:

Message_At (ModeReceiveInterface)	T
ModeReceiveData = ModeType::kWayPointThrough	T

:= cWayPointLand

Condition:

Message_At (ModeReceiveInterface)	T
ModeReceiveData = ModeType::kWayPointLand	T

:= cPilotAssist

Condition:

Message_At (ModeReceiveInterface)	T
ModeReceiveData = ModeType::kPilotAssist	T

Constant

Delay

Type: TIME

Value: 2 S

Constant

RadOfCapture

Type: INTEGER

Units: ft

Value: 40

Macro

WithInRadOfCapture

Parameters: NONE

Condition:

(WPInXVar – LWPInXVar) > - RadOfCapture	T
RadOfCapture > (WPInXVar – LWPInXVar)	T
(WPInYVar – LWPInYVar) > - RadOfCapture	T
RadOfCapture > (WPInYVar – LWPInYVar)	T
(WPInZVar – LWPInZVar) > - RadOfCapture	T
RadOfCapture > (WPInZVar – LWPInZVar)	T

APPENDIX C – Executable Specification

```
/*
 *
 * COMPONENT FMS0512
 *
 * Converted by NimbusConvert version 1.0
 * Copyright 1999, SAFEWARE CORP.
 * SOURCE FILE:
D:\programs\apache2\htdocs\projects\Safeware\April\FMS0512.doc
 * CONVERSION DATE: May 12, 2000, 14:26:58
 *
 */

COMPONENT FMS0512 :

/*
 *
 * Type Definitions
 *
 */

TYPE_DEF ModeType {kGroundMode, kRunUp, kTakeOff, kWayPointThrough,
kWayPointHover, kWayPointLand, kPilotAssist, kUnknown }

/*
 *
 * Messages
 *
 */

MESSAGE NewWPTriggerMessage {
   Trigger IS INTEGER
}

MESSAGE ModeOutputMessage {
   WPOutX IS INTEGER,
   WPOutY IS INTEGER,
   WPOutZ IS INTEGER,
   MOD IS ModeType
}

MESSAGE InitializeMessage {}

MESSAGE ModeReceiveMessage {
   WPInX IS INTEGER,
   WPInY IS INTEGER,
   WPInZ IS INTEGER,
   MRD IS ModeType
}

MESSAGE LiveStatusMessage {
```

```
   LWPInX IS INTEGER,
   LWPInY IS INTEGER,
   LWPInZ IS INTEGER,
   LMRD IS ModeType
}

/*
 *
 * State Hierarchy
 *
 */

STATE FMS_Controller :
   EQUIVALENCE FMS :
      STATE FMS_Dummy DEFAULT :
      /*
The Dummy had to be used as a transient state
*/
EQUIVALENCE OperatingStatus :
ATOMIC TaskList :
ATOMIC ReturnHome :
ATOMIC Manual :
ATOMIC Off DEFAULT :
ATOMIC Initializing :
END EQUIVALENCE
END STATE
END EQUIVALENCE
EQUIVALENCE HelicopterModel :

STATE HelicopterState DEFAULT :
EQUIVALENCE PerceivedMode :
ATOMIC GroundMode DEFAULT :
ATOMIC RunUp:
ATOMIC TakeOff :
ATOMIC WayPointThrough :
ATOMIC WayPointHover :
ATOMIC WayPointLand :
ATOMIC PilotAssist :
ATOMIC Unknown :
END EQUIVALENCE
EQUIVALENCE EMCS :
ATOMIC Acknowledged :
ATOMIC Unacknowledged DEFAULT :
END EQUIVALENCE
EQUIVALENCE EstimatedFlightModeStatus :
ATOMIC Incomplete :
ATOMIC Complete :
ATOMIC Unknown DEFAULT :
END EQUIVALENCE
EQUIVALENCE EstimatedGPSStatus :
ATOMIC OKAY :
ATOMIC DOWN :
ATOMIC Unknown DEFAULT :
END EQUIVALENCE
EQUIVALENCE EstimatedSonarStatus :
ATOMIC OKAY :
ATOMIC DOWN :
```

```
ATOMIC Unknown DEFAULT :
END EQUIVALENCE
EQUIVALENCE EstimatedNavFilterStatus :
ATOMIC Healthy :
ATOMIC Standby :
ATOMIC Down :
ATOMIC Unknown DEFAULT :
END EQUIVALENCE
EQUIVALENCE EstimatedAltitude :
ATOMIC AboveGLFiveFeet :
ATOMIC AboveGLOneFeet :
ATOMIC OnGround :
ATOMIC Unknown DEFAULT :
END EQUIVALENCE
EQUIVALENCE CommandedModeChange :
ATOMIC cGroundMode :
ATOMIC cRunUp:
ATOMIC cTakeOff :
ATOMIC cWayPointThrough :
ATOMIC cWayPointHover :
ATOMIC cWayPointLand :
ATOMIC cPilotAssist :
ATOMIC cUnknown DEFAULT :
END EQUIVALENCE
END STATE
END EQUIVALENCE
END STATE

/*
 *
 * Constants
 *
 */

CONSTANT Delay : TIME
   VALUE : 2 S
END CONSTANT

CONSTANT RadOfCapture : INTEGER
   UNITS : ft
   VALUE : 40
END CONSTANT

/*
 *
 * Input Variables
 *
 */

IN_VARIABLE ModeReceiveData : ModeType
   INITIAL_VALUE : Undefined
END IN_VARIABLE

IN_VARIABLE WPInXVar : INTEGER
   UNITS : meters
   INITIAL_VALUE : Undefined
   EXPECTED_MIN : 0
```

```
   EXPECTED_MAX : 10000
END IN_VARIABLE

IN_VARIABLE WPInYVar : INTEGER
   UNITS : meters
   INITIAL_VALUE : Undefined
   EXPECTED_MIN : 0
   EXPECTED_MAX : 10000
END IN_VARIABLE

IN_VARIABLE WPInZVar : INTEGER
   UNITS : meters
   INITIAL_VALUE : Undefined
   EXPECTED_MIN : 0
   EXPECTED_MAX : 10000
END IN_VARIABLE

IN_VARIABLE LiveModeData : ModeType
   INITIAL_VALUE : Undefined
END IN_VARIABLE

IN_VARIABLE LWPInXVar : INTEGER
   UNITS : meters
   INITIAL_VALUE : Undefined
   EXPECTED_MIN : 0
   EXPECTED_MAX : 10000
END IN_VARIABLE

IN_VARIABLE LWPInYVar : INTEGER
   UNITS : meters
   INITIAL_VALUE : Undefined
   EXPECTED_MIN : 0
   EXPECTED_MAX : 10000
END IN_VARIABLE

IN_VARIABLE LWPInZVar : INTEGER
   UNITS : meters
   INITIAL_VALUE : Undefined
   EXPECTED_MIN : 0
   EXPECTED_MAX : 10000
END IN_VARIABLE

/*
 *
 * Output Variables
 *
 */

OUT_VARIABLE ModeOutputVariable : ModeType
   ASSIGNMENT :
      EQUALS ModeType::kGroundMode IF
         TABLE
            ..CommandedModeChange IN_STATE cGroundMode : T ;
         END TABLE;
      EQUALS ModeType::kRunUp IF
         TABLE
            ..CommandedModeChange IN_STATE cRunUp : T ;
```

```
         END TABLE;
      EQUALS ModeType::kTakeOff IF
         TABLE
            ..CommandedModeChange IN_STATE cTakeOff : T ;
         END TABLE;
      EQUALS ModeType::kWayPointThrough IF
         TABLE
            ..CommandedModeChange IN_STATE cWayPointThrough : T ;
         END TABLE;
      EQUALS ModeType::kWayPointHover IF
         TABLE
            ..CommandedModeChange IN_STATE cWayPointHover : T ;
         END TABLE;
      EQUALS ModeType::kWayPointLand IF
         TABLE
            ..CommandedModeChange IN_STATE cWayPointLand : T ;
         END TABLE;
      EQUALS ModeType::kPilotAssist IF
         TABLE
            ..CommandedModeChange IN_STATE cPilotAssist : T ;
         END TABLE;
      EQUALS ModeType::kUnknown IF
         TABLE
            ..CommandedModeChange IN_STATE cUnknown : T ;
         END TABLE;
END OUT_VARIABLE

/*
 *
 * Macros
 *
 */

MACRO WithInRadOfCapture() :
   TABLE
      (WPInXVar - LWPInXVar) > - RadOfCapture : T ;
      RadOfCapture > (WPInXVar - LWPInXVar) : T ;
      (WPInYVar - LWPInYVar) > - RadOfCapture : T ;
      RadOfCapture > (WPInYVar - LWPInYVar) : T ;
      (WPInZVar - LWPInZVar) > - RadOfCapture : T ;
      RadOfCapture > (WPInZVar - LWPInZVar) : T ;
   END TABLE
END MACRO

/*
 *
 * State Variable: OperatingStatus
 *
 */

TRANSITION Off TO Initializing :
   LOCATION : FMS_Controller.FMS.FMS_Dummy.OperatingStatus
   CONDITION :
      TABLE
         MESSAGE_AT(InitializeSystemInterface) : T ;
      END TABLE
END TRANSITION
```

```
/*
 *
 * State Variable: EstimatedFlightModeStatus
 *
 */

TRANSITION ANY TO Complete :
   LOCATION :
FMS_Controller.HelicopterModel.HelicopterState.EstimatedFlightModeStatus
   CONDITION :
      TABLE
         WithInRadOfCapture() : T ;
      END TABLE
END TRANSITION

TRANSITION ANY TO Incomplete :
   LOCATION :
FMS_Controller.HelicopterModel.HelicopterState.EstimatedFlightModeStatus
   CONDITION :
      TABLE
         WithInRadOfCapture() : F ;
      END TABLE
END TRANSITION

/*
 *
 * State Variable: PerceivedMode
 *
 */

TRANSITION ANY TO GroundMode :
   LOCATION :
FMS_Controller.HelicopterModel.HelicopterState.PerceivedMode
   CONDITION :
      TABLE
         Message_At (LiveStatusInterface) : T ;
         LiveModeData = ModeType::kGroundMode : T ;
      END TABLE
END TRANSITION

TRANSITION ANY TO RunUp :
   LOCATION :
FMS_Controller.HelicopterModel.HelicopterState.PerceivedMode
   CONDITION :
      TABLE
         Message_At (LiveStatusInterface) : T ;
         LiveModeData = ModeType::kRunUp : T ;
      END TABLE
END TRANSITION

TRANSITION ANY TO TakeOff :
   LOCATION :
FMS_Controller.HelicopterModel.HelicopterState.PerceivedMode
   CONDITION :
      TABLE
         Message_At (LiveStatusInterface) : T ;
```

```
         LiveModeData = ModeType:: kTakeOff : T ;
      END TABLE
END TRANSITION

TRANSITION ANY TO WayPointHover :
   LOCATION :
FMS_Controller.HelicopterModel.HelicopterState.PerceivedMode
   CONDITION :
      TABLE
         Message_At (LiveStatusInterface) : T ;
         LiveModeData = ModeType::kWayPointHover : T ;
      END TABLE
END TRANSITION

TRANSITION ANY TO WayPointThrough :
   LOCATION :
FMS_Controller.HelicopterModel.HelicopterState.PerceivedMode
   CONDITION :
      TABLE
         Message_At (LiveStatusInterface) : T ;
         LiveModeData = ModeType::kWayPointThrough : T ;
      END TABLE
END TRANSITION

TRANSITION ANY TO WayPointLand :
   LOCATION :
FMS_Controller.HelicopterModel.HelicopterState.PerceivedMode
   CONDITION :
      TABLE
         Message_At (LiveStatusInterface) : T ;
         LiveModeData = ModeType::kWayPointLand : T ;
      END TABLE
END TRANSITION

TRANSITION ANY TO PilotAssist :
   LOCATION :
FMS_Controller.HelicopterModel.HelicopterState.PerceivedMode
   CONDITION :
      TABLE
         Message_At (LiveStatusInterface) : T ;
         LiveModeData = ModeType::kPilotAssist : T ;
      END TABLE
END TRANSITION

/*
 *
 * State Variable: CommandedModeChange
 *
 */

TRANSITION ANY TO cGroundMode :
   LOCATION :
FMS_Controller.HelicopterModel.HelicopterState.CommandedModeChange
   CONDITION :
      TABLE
         Message_At (ModeReceiveInterface) : T ;
         ModeReceiveData = ModeType::kGroundMode : T ;
```

```
      END TABLE
END TRANSITION

TRANSITION ANY TO cRunUp :
   LOCATION :
FMS_Controller.HelicopterModel.HelicopterState.CommandedModeChange
   CONDITION :
      TABLE
         Message_At (ModeReceiveInterface) : T ;
         ModeReceiveData = ModeType::kRunUp : T ;
      END TABLE
END TRANSITION

TRANSITION ANY TO cTakeOff :
   LOCATION :
FMS_Controller.HelicopterModel.HelicopterState.CommandedModeChange
   CONDITION :
      TABLE
         Message_At (ModeReceiveInterface) : T ;
         ModeReceiveData = ModeType:: kTakeOff : T ;
      END TABLE
END TRANSITION

TRANSITION ANY TO cWayPointHover :
   LOCATION :
FMS_Controller.HelicopterModel.HelicopterState.CommandedModeChange
   CONDITION :
      TABLE
         Message_At (ModeReceiveInterface) : T ;
         ModeReceiveData = ModeType::kWayPointHover : T ;
      END TABLE
END TRANSITION

TRANSITION ANY TO cWayPointThrough :
   LOCATION :
FMS_Controller.HelicopterModel.HelicopterState.CommandedModeChange
   CONDITION :
      TABLE
         Message_At (ModeReceiveInterface) : T ;
         ModeReceiveData = ModeType::kWayPointThrough : T ;
      END TABLE
END TRANSITION

TRANSITION ANY TO cWayPointLand :
   LOCATION :
FMS_Controller.HelicopterModel.HelicopterState.CommandedModeChange
   CONDITION :
      TABLE
         Message_At (ModeReceiveInterface) : T ;
         ModeReceiveData = ModeType::kWayPointLand : T ;
      END TABLE
END TRANSITION

TRANSITION ANY TO cPilotAssist :
   LOCATION :
FMS_Controller.HelicopterModel.HelicopterState.CommandedModeChange
   CONDITION :
```

```
      TABLE
         Message_At (ModeReceiveInterface) : T ;
         ModeReceiveData = ModeType::kPilotAssist : T ;
      END TABLE
END TRANSITION

/*
 *
 * Interfaces
 *
 */

OUT_INTERFACE NewWPTriggerInterface :
   CHANNEL : NewWPTriggerChannel
   MIN_SEP : 100 MS
   MAX_SEP : 200 MS
   OUTPUT_ACTION : SEND(NewWPTriggerMessage)
   HANDLER :
      CONDITION :
         TABLE
            ..EstimatedFlightModeStatus IN_STATE Complete : T ;
            PREV_STEP(..EstimatedFlightModeStatus IN_STATE Incomplete) :
T ;
         END TABLE
      ASSIGNMENT
         Trigger := 1
      END ASSIGNMENT
      ACTION : NONE
   END HANDLER
END OUT_INTERFACE

OUT_INTERFACE ModeOutputInterface :
   CHANNEL : ModeOutputChannel
   MIN_SEP : 100 MS
   MAX_SEP : 200 MS
   OUTPUT_ACTION : SEND(ModeOutputMessage)
   HANDLER :
      CONDITION :
         TABLE
            PREV_STEP(ModeOutputVariable) != ModeOutputVariable : T ;
         END TABLE
      ASSIGNMENT
         WPOutX := WPInXVar,
         WPOutY := WPInYVar,
         WPOutZ := WPInZVar,
         MOD := ModeOutputVariable
      END ASSIGNMENT
      ACTION : NONE
   END HANDLER
END OUT_INTERFACE

IN_INTERFACE InitializeSystemInterface :
   CHANNEL : InitializeChannel
   MIN_SEP : 50 MS
   MAX_SEP : 100 MS
   INPUT_ACTION : RECEIVE(InitializeMessage)
   RECEIVE_HANDLER :
```

```
      CONDITION : TRUE
   END HANDLER
END IN_INTERFACE

IN_INTERFACE ModeReceiveInterface :
   CHANNEL : ModeInputChannel
   MIN_SEP : 50 MS
   MAX_SEP : 100 MS
   INPUT_ACTION : RECEIVE(ModeReceiveMessage)
   RECEIVE_HANDLER :
      CONDITION : TRUE
      ASSIGNMENT
         WPInXVar := WPInX,
         WPInYVar := WPInY,
         WPInZVar := WPInZ,
         ModeReceiveData := MRD
      END ASSIGNMENT
   END HANDLER
END IN_INTERFACE

IN_INTERFACE LiveStatusInterface :
   CHANNEL : LiveChannel
   MIN_SEP : 50 MS
   MAX_SEP : 100 MS
   INPUT_ACTION : RECEIVE(LiveStatusMessage)
   RECEIVE_HANDLER :
      CONDITION : TRUE
      ASSIGNMENT
         LWPInXVar := LWPInX,
         LWPInYVar := LWPInY,
         LWPInZVar := LWPInZ,
         LiveModeData := LMRD
      END ASSIGNMENT
   END HANDLER
END IN_INTERFACE

END COMPONENT
```