Onboard Agents for Autonomous Operation of Micro Satellites

Deepak Mohan

School of Electrical and Electronics Engineering

A thesis submitted to the Nanyang Technological University
in fulfilment of the requirement for the degree of
Master of Engineering

2006

Acknowledgements

First and foremost, I would like to express my most sincere gratitude towards my supervisor Assoc/Prof. K Arichandran. The guidance and inspiration he provided through the course of this project has been invaluable and the project would not have progressed to its current state without his supervision.

I would like to take this opportunity to thank Assoc/Prof. Tan Soon Hie, the Director of the Satellite Engineering Centre where this Masters work was carried out. I am also grateful to Dr Ian McLoughlin, for his technical advice and support during the development.

Last but not the least, I would like to express my gratitude and appreciation to the staff and students at the Satellite Engineering Centre, for the technical help, encouragement and support they have given me. Particular mention must be made of Mr K L Tan, Mr Gurprakash Singh, Mr Shantanu Shukla and Mr Manuj Prakash for their help at various points of time in this project.

Abstract

Traditionally, satellite missions are operated with a lot of interaction with the ground station on earth. This however is slowly changing and there is an increasing interest in the field of autonomous satellite operations. The work done in this project started with a study of the necessary capabilities for an autonomous micro-satellite platform. Issues of hardware fault tolerance and survival were also considered.

Based on this a baseline architecture to achieve incremental layers of autonomy is presented. This architecture enables evaluation and testing autonomous operations on actual micro-satellite missions. A layered approach is used where layers of increasing complexity are built in a bottom up fashion, adding on higher functional capabilities to the system. The guiding factors for the design are cost-efficiency, reliability and scalability.

Prototype hardware and functionalities of baseline flight software applications, implemented based on VxWorks operating system on an ERC-32 processor, were developed and evaluated in the laboratory. The design for a multiple agent based autonomous operations software layer for mission-planning and fault handling, that would run on top of this platform, is presented. The mission-planning scheme is based on a STRIPS like planner working on a logical satellite model. The mission planning operation was also assessed in a PC based LISP interpreter environment under simulated typical scenarios to gain an understanding of the complexity and efficiency of the planner.

The research and development done here give an understanding of the concept of autonomous operations for micro-satellites, the various functionalities required to achieve this and a possible approach to develop an autonomous micro-satellite platform.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS II

ABSTRACT III

TABLE OF CONTENTS IV

LIST OF FIGURES VIII

LIST OF TABLES X

LIST OF ACRONYMS XI

1. INTRODUCTION 1

1.1 MOTIVATION 2
1.2 BACKGROUND 3
1.3 PROBLEM STATEMENT 4
1.4 TARGET SYSTEM AND ENVIRONMENT: THE X-SAT PROTOTYPE MODEL 4
1.5 ON-BOARD AUTONOMY - FAULT TOLERANCE AND AI 5
1.6 OVERVIEW OF FLIGHT COMPUTER ARCHITECTURE 7
1.6.1 Flight Computer Hardware 8
1.6.2 Platform Software 9
1.6.3 Baseline Operations Software 11
1.6.4 Autonomous Mission Operations Layer 11
1.7 ORGANISATION OF THESIS 13

2. BACKGROUND AND APPROACHES 14

2.1 OVERVIEW 15
2.2 HARDWARE PERSPECTIVE 15
2.2.1 Space Environment and Electronics 15
2.2.2 Design Approaches for Reliability 16
2.3 SOFTWARE PERSPECTIVE 18
2.3.1 Roadmap of AI Development 18
2.3.2 Agents 23
2.4 AUTONOMOUS SPACE MISSIONS 25

2.4.1 DeepSpace-1 ..... *25*
2.4.2 PROBA *27*
2.4.3 TechSat 21 – Autonomous Science Experiment (ASE) *29*
2.5 LAYERED ARCHITECTURE 31
2.5.1 Key Features *32*
3. FAULT TOLERANT HARDWARE PLATFORM 34
3.1 OVERVIEW 35
3.2 OBC HARDWARE DESIGN 36
3.3 FPGA FUNCTIONALITY 44
4. BASELINE OPERATIONS SOFTWARE 46
4.1 OVERVIEW 47
4.2 SOFTWARE PLATFORM 48
4.2.1 Operation *49*
4.3 TELEMETRY COLLECTION AND MONITORING 51
4.3.1 Operation *51*
4.3.2 Interfaces *52*
4.4 COMMAND EXECUTION 53
4.4.1 Operation *53*
4.4.2 Safety Layer *55*
4.4.3 Interfaces *55*
4.5 GROUND INTERFACE 56
4.5.1 Operation *56*
4.5.2 Interface to autonomous operations modules *57*
5. HIGH LEVEL AUTONOMY FOR MICRO-SATELLITE OPERATIONS 58
5.1 PARADIGM FOR OPERATIONAL AUTONOMY 59
5.1.1 The Environment *59*
5.1.2 The Operations and Mission Goals *61*
5.1.3 Information Integrity *63*
5.1.4 Some Considerations for Higher Levels of Autonomy in Operations .. *63*

5.2 APPROACH FOR AUTONOMY 66
5.2.1 Schemes for autonomous mission planning *67*
5.2.2 Knowledge Representation and Logic based Planning *67*
5.2.3 Propositional Logic *68*
5.2.4 First Order Predicate Logic *70*
5.2.5 Planning using Situational Calculus *72*
5.2.6 The STRIPS Planning Language *73*
5.2.7 State Update & Data Integrity Verification *74*
5.2.8 Scheduling *74*
5.3 FAULT DIAGNOSIS 74
5.3.1 Possible Schemes *75*

6. AUTONOMOUS OPERATIONS SOFTWARE 79

6.1 OVERVIEW 80
6.2 MISSION PLANNER AGENT 82
6.2.1 The STRIPS Scheme *82*
6.2.2 The Flight Software Internal Knowledge Representation *85*
6.2.3 Mission Planning *88*
6.2.4 Plan Release *96*
6.3 STATE MAINTENANCE AGENT 96
6.3.1 Mapping Real World Information to Model *96*
6.3.2 The Verification Database *98*
6.3.3 Update of Verification Database *100*
6.4 FAULT HANDLER AGENT 102
6.4.1 Overview *102*
6.4.2 Classification *103*
6.4.3 Action *105*
6.4.4 Confirmation *105*

7. CONCLUSION 107

7.1 RESULTS 108
7.1.1 Hardware Platform and Baseline Operations *108*

7.1.2 Evaluation of Autonomous Mission Planner *109*

7.2 CONCLUSION AND FUTURE WORK 114

7.3 AI AND SPACE 116

REFERENCES **118**

APPENDIX A – X-SAT BUS AND PAYLOADS **1**

APPENDIX B – HARDWARE SCHEMATICS **1**

APPENDIX C –MISSION PLANNER KNOWLEDGE BASE (LISP) **1**

LIST OF FIGURES

Figure 1 Flight Computer Block Diagram 8
Figure 2 Tasks launched by the Task Manager 10
Figure 3 Autonomous Agents in Flight Computer 12
Figure 4 A Simple Perceptron 21
Figure 5 Multi layered Perceptron from simple units 22
Figure 6 Taxonomy of autonomous agents 24
Figure 7 Collaborative Agents 24
Figure 8 DS-1 Remote Agent Architecture 25
Figure 9 Typical planning vs Casper's iterative planning approach 28
Figure 10 TechSat-21 Flight Software Layers [16] 30
Figure 11 Multi-layered system in operating environment 31
Figure 12 Hardware Platform in the context of the Flight Computer Layers 35
Figure 13 Block Diagram of On Board Computer Hardware 36
Figure 14 ERC-32 Processor Block 37
Figure 15 FPGA Block 37
Figure 16 SRAM Banks Block 39
Figure 17 Flash Banks Block 39
Figure 18 Current limiting circuit 40
Figure 19 Power Input Block 41
Figure 20 CAN bus Interface 41
Figure 21 FPGA Communication links and ERC-32 45
Figure 22 Baseline Flight Software in the context of the Flight Computer Layers 47
Figure 23 Power up sequence 50
Figure 24 Flight Computer and its Operating Environment 60
Figure 25 Typical Micro-satellite Environment 61
Figure 26 Ground-satellite interactions for normal vs autonomous missions 62
Figure 27 Agent uses KB or Model to reason about environment 68
Figure 28 Structure of a Rule based expert system 76
Figure 29 Autonomous Operation Layer in the Context of the Flight Computer Layers 80

Figure 30 Modules for autonomous operation 81
Figure 31 STRIPS Based Planner 85
Figure 32 OBC Initialisation 87
Figure 33 Different Modes of Operation (States) of the OBC 88
Figure 34 Sample Area of Interest Database 89
Figure 35 Fault Handling Operations 106
Figure 36 Prototype Platform of Flight Computer 108
Figure 37 Flowchart of Activities for Autonomous Operation 110
Figure 38 Block diagram of planner used for evaluation 112
Figure 39 Plans generated by the mission planner for various mission goals ... 113
Figure 40 X-Sat Sub-Systems and Connectivity 2

LIST OF TABLES

Table 1 Telemetry Collection LUT 52
Table 2 Propositional logic syntax 69
Table 3 Propositional logic semantics 69
Table 4 FOL syntax 71
Table 5 Predicates used in the model representing the satellite 86
Table 6 Objects in the model representing the satellite 87
Table 7 Sample Goal List 90
Table 8 Goal macros and STRIPS goal sequence expansions 92
Table 9 Action Schema List 94
Table 10 Mapping of predicates to TM / Status inputs to OBC 98
Table 11Generic verification database for current / temperature telemetry and command effects 100
Table 12 State Monitoring and Specification Update 101
Table 13 Sample entries for the fault models database 104
Table 14 Sample fault action plans 105
Table 15 Action paths searched during planning 114

List of Acronyms

ADAM	Advanced Data Acquisition and Messaging
ADCS	Attitude Determination and Control Services
AI	Artificial Intelligence
API	Application Programming Interface
ASE	Autonomous Sciencecraft Experiment
ATCP	Absolute Time Command Processor
CAN	Controller Area Network
CASPER	Continuous Activity Scheduling Planning Executing and Replanning
CCSDS	Consultative Committee for Space Data Systems
DS-1	Deep Space 1
EDAC	Error Detection and Correction
EEPROM	Electrically-Erasable Programmable Read-Only Memory
ESA	European Space Agency
FPGA	Field Programmable Gate Array
GEO	Geosynchronous Earth orbit
GPS	Global Positioning Systems
GPS (A)	GPS Receiver from Accord Systems
GPS (P)	Phoenix GPS Receiver from DLR, Germany
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
JTAG	Joint Test Action Group of IEEE
LEO	Low Earth Orbit
LISP	List Processing
LUT	Look Up Table
MB	Mega Bytes
MI-R	Identification and Re-configuration
MT	Mobile Terminal
NASA	National Aeronautics and Space Administration

NTU	Nanyang Technological University
OBC	On-Board Computer
PC	Personal Computer
PCB	Printed Circuit Board
PPS	Pulse Per Second
PPU	Parallel Processing Unit
PSU	Power Supply Unit
RF	Radio Frequency
SCL	Spacecraft Command Language
SEC	Satellite Engineering Centre
SEE	Single Event Effect
SEL	Single Event Latchup
SEU	Single Event Upset
SPARC	Scalable Processor Architecture
SRAM	Static Random Access Memory
STRIPS	STanford Research Institute Problem Solver
TID	Total Ionising Dose
TT &C	Telemetry, Tracking and Command
TX	Transmit
UART	Universal Asynchronous Receiver/Transmitter
WOD	Whole Orbit Data

1. Introduction

1.1 Motivation

Typically micro satellites are controlled with a high amount of interaction with the ground station on earth. This is mainly due to the high cost of the mission and the need for reliability. The ground stations handling the satellite require to be manned by trained staff throughout the life of the satellite. This ensures higher reliability but with high operational costs. Besides, the satellite is not able to optimise payload operations as schedules are planned and fixed much ahead of opportunity window.

On board autonomy on such missions would help reduce operational costs on the ground as well as maximise the utility of the payloads on board. Another important reason to support autonomous operation is to prepare for future exploratory missions that may have to be unmanned and may be out of human operator contact for extended periods of time. Several of the leading groups in the field, including NASA and ESA, are putting research and development effort into this field. Design of a scalable autonomous computer system is a necessary step towards the future of space missions.

Transitioning to entirely autonomous missions would require considerable amount of development, testing and validation effort, before it becomes an attractive and viable option for the space community. A useful approach would be to design missions that support modes of independent operation with minimum ground intervention, triggered by ground command, to enable validation of autonomous operations. Thus, the mission would be able to be carried out in the typical manner, using ground commands scheduling each activity. When considered appropriate, the ground station would be able to trigger initialisation of software modules required for autonomous mission operations, which would then manage the mission for a certain period of time.

This approach would help build confidence and heritage in the design, at the same time supporting typical operations in the classical fashion and maintaining a high level of reliability.

1.2 Background

Micro-satellites are satellites that fall in the 10 – 100kg category within the small satellite classification. They are used for a wide range of applications including communications, imaging and scientific experimentation and observations. Based on their capabilities, micro-satellites can be divided into two distinct classes, 'traditional' and 'modern' [1]. The key difference between the two are the availability of an on board computer on board modern micro-satellites. This in turn provides higher processing capabilities and functionality, even out of range of the ground station, on board these micro-satellites.

Traditional micro-satellites include the earliest satellites to be launched including the Sputnik-1 and Explorer-1 missions. The ground station to satellite interface on these are limited to only low-level commands for switching and configuring of the system. Higher level operations and scheduled in orbit payload operations out of orbit were not easily possible. Modern micro-satellite missions with on board processing capability started in the early 1980s with the UoSat-1 [2]. The availability of an on board computer greatly increased the capability of these missions.

Worldwide interest in Low Earth Orbit (LEO) micro-satellites has shown a great increase in the last decade. Some of the features that make micro-satellites attractive are their low cost of development, relatively shorter development time and lower cost of launch, as compared to bigger satellites [3]. In recent years, an approach that has greatly contributed to this is to capitalise on the use of Components off the Shelf (COTS) electronics devices instead of special space qualified devices for space missions [4].

Unlike geostationary satellites, which are always in contact with the ground station, LEO have only small duration of ground contact per orbit. Typical operation of a micro-satellite mission involves a polar orbit of a few (around 4) ground station contact periods (ground station at latitude 0 degrees, and satellite in polar orbit at an altitude of 700km.) that last around 10 minutes each every day. During these slots, the ground station collects from the satellite telemetry information about its status, events in the last few orbits . The data could potentially be used for diagnosis of problems in the spacecraft. In the latter

case, ground station personnel have to assess the problems that arise, decide on the corrective actions and upload commands to effect corrective action. The satellite follows these schedules and carries out operations as specified. Faults or anomalies are logged as telemetry for the ground to diagnose and take action.

1.3 Problem Statement

The work done in this research is to build from bottom up, a fault tolerant computing hardware; and design the flight software to enable on board autonomy for LEO micro-satellite missions. This would be a cost efficient hardware platform, minimising the use of space grade components and employing design features such as redundancy and modularity to increase reliability. The flight software design would include software needed for baseline operations as well as the modules for autonomous operation. As mentioned earlier, this would make it possible to validate autonomous operations and enable a gradual transition to a completely autonomous mission.

The target framework used for the design is the X-Sat micro-satellite, under development at the Satellite Engineering Centre in NTU. However, the system is intended to be scalable, modular and reusable to meet varying mission and reliability requirements. Prototypes developed would be tested for functionality and fault tolerance through simulated interactions. If feasible, the system would be integrated with available prototypes of other sub-systems and tested.

The functionalities needed for autonomy would be designed as software agents, which can be initiated during the mission through ground commands. These agents would run along side the core flight software in a decoupled manner and generate commands required to achieve ground assigned mission goals, thus making the autonomous operation transparent to the baseline software and the rest of the system.

1.4 Target System and Environment: The X-Sat Prototype Model

The platform for which the design and development of the autonomous flight computer is the prototype model of the X-Sat micro-satellite. The design of the computer hardware

and software is based on the requirements of the X-Sat mission. A brief summary is given in Appendix A.

X-Sat Flight Computer: The main functionalities required of the X-Sat Flight computer, the On Board Computer (OBC) are:

i. Telemetry collection & monitoring: Collection, logging and real time download during ground station pass of various telemetry data generated on board.
ii. Command execution: Execution of immediate and time-scheduled commands.
iii. Ground communication: Exchange telemetry, commands and data with the ground station over the Telemetry Tracking and Command (TT&C) link.

Besides the functional requirements, the computer is also required to be able to tolerate any internal single point failure. No single error must result in total failure.

Autonomous Operations: The following are the autonomous functionalities targeted for the system.

i. Fault diagnosis and recovery
ii. Scheduling of mission activities based on high level goals set by the ground

1.5 On-Board Autonomy - Fault Tolerance and AI

One of the reasons for limited autonomy on earlier micro satellites was the limited availability of computing power on board. With the availability of higher processing power microprocessors for micro-satellite, it is possible for software with a higher level of functionality and autonomy to reside on board.

The utilisation of non-space grade devices for micro satellite missions makes it necessary to first ensure that the operating platform is capable of tolerating potential faults and operating in a fail-safe manner. The on board control software will have to sense the environment from the telemetry available, both within the satellite and outside, and handle activity that would ensure that all transactions are fulfilled. The key features required for such operations include self-checking to verify satellite integrity and scheduling of activities for the best possible results.

In order to efficiently do this, AI techniques that help increase the capability of the software will need to be employed. Scalability is an important feature here. This ensures that the system can be implemented and tested in a progressive and reliable manner. To support scalability and mission dependent re-configuration, a certain level of modularity and decoupling between modules is also essential. This can be done by categorising similar functionalities together into design unit, and abstracting their internal implementations away from the rest of the system. The interaction between the design units should be only through the functional interfaces provided by the units.

The On Board Data Handling (OBDH) sub-system X-Sat is designed to have an On Board Computer (OBC) for processing, a RamDisk for mass data storage and a distributed Controller Area Network (CAN) bus network for communications. Each system interfaces to the CAN bus through a CAN node. The CAN node consists of a c515c microprocessor supporting the sub-system specific interface on one side and the CAN bus interface on the other side. Thus, besides the OBC, the X-Sat bus has at least 4 microprocessors connected together. With the available computational resources on board, the following scenarios are possible to support capability for autonomous operations.

Scenario 1 is a centralised system with the flight computer scheduling all autonomous operations. The CAN node computers on the bus are kept in a suppressed mode to respond only to requests and commands form the main flight computer in a master slave manner. The flight software would receive high-level mission goals from the ground and schedule the activities required to achieve these goals autonomously. At the appropriate times, the main flight computer sends the required CAN commands to the various CAN nodes.

Scenario 2 would be a distributed control network consisting of the flight computer and all the CAN node computers on the bus. Each node would be responsible for scheduling activities on the sub-system connected to it. Synchronisation of activities and plans over

the CAN bus would be necessary to ensure successful operation of the system. The TT&C CAN node would be responsible for receiving high level goals from the ground and relaying it to all the other nodes. A distributed planning scheme would then be used to plan locally, resolve resource or schedule constraints and execute the plan.

The design proposed in this thesis is based on the centralised approach of Scenario 1. During normal operation, the system is planned to be operated in a master slave manner, through commands from the ground, routed through the flight computer. One of the main advantages Scenario 1 has is that the switch from typical operation to autonomous operation affects only one unit, the flight software. The change in operation method is transparent to the rest of the sub-systems. Although it is possible for the CAN nodes to send requests and telemetry data asynchronously, the bus is operated in a master-slave manner by the flight computer. This is done to limit the rate of arrival of CAN messages and avoid data congestion at the flight computer. A centralised system for on-board autonomy, residing in the flight computer, would be designed to carry out mission with a minimal amount of interaction with the ground.

The higher-level autonomy is intended to be autonomous software agents, running along side the core flight software, which carry out dedicated tasks to satisfy specific requirements. Maintaining these routines as external agents abstracts this operation from the rest of the mission activities of the satellite. This abstraction also partitions the autonomous operation from the baseline flight software, ensuring mutual decoupling, error isolation and modularity.

1.6 Overview of Flight Computer Architecture

The flight computer consists of four distinct layers. These are the hardware, the platform software, the baseline operations software and the autonomous mission operation layers. Figure 1 gives a block diagram of the various layers of the flight computer.

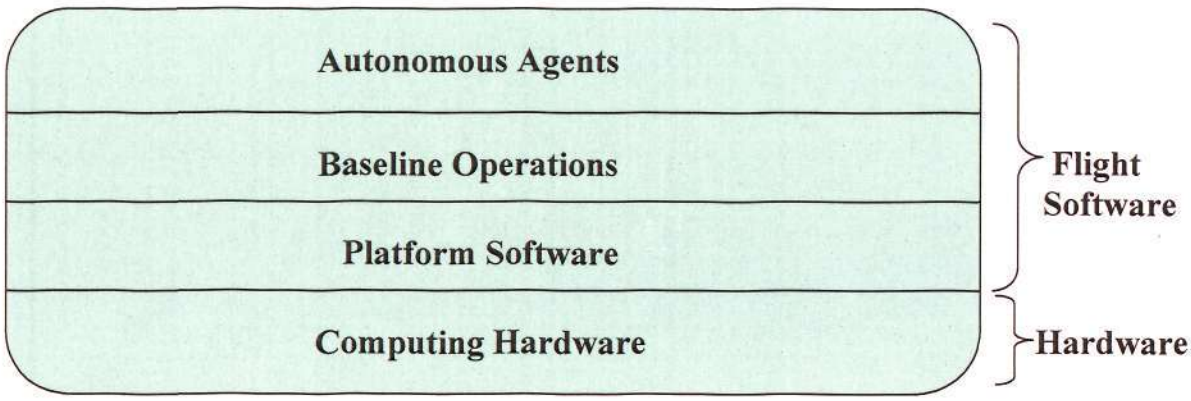

Figure 1 Flight Computer Block Diagram

1.6.1 Flight Computer Hardware

The flight computer hardware would provide the computing resources required to host and carry out the operations of the flight software. The functional requirements on the computing hardware are the processing speed, run-time memory (RAM), non-volatile memory and interfaces required of the flight computer. The satellite is expected to go through high levels of mechanical stress (during the launch phase) and exposure to radiation levels much higher than that experienced on the ground during orbit. The radiation is particularly bad for semi-conductor devices on the satellite. These may produce potentially fatal errors in various devices in the system. Thus, the design must also support fault tolerance features to overcome errors and enable it to operate robustly in the harsh space environment.

One of the approaches to increase the reliability of computing systems in space is to employ specially fabricated and tested "space grade" devices for all the functionalities on the board. These devices are usually much more expensive and usually have considerably higher power consumption, mass and size compared to the their non space-grade counterparts. This however is not a very cost efficient approach, especially for micro-satellite missions where cost, space and power constraints are much tighter. Besides, space-grade devices are only available for a very limited set of functionalities. Restricting the design to the usage of available space-grade devices severely narrows the variety of functionalities and options available for the system.

The alternative is to employ non space-grade devices for the design. This is particularly true for Low Earth Orbit (LEO) missions where the radiation levels and effect on electronics device is not as high as MEO or GEO missions. The reliability level of such systems can be increased through

i. Careful selection of parts, based on heritage, available radiation tests and inherent characteristics of the device; and
ii. Design strategies such as redundancy, Error Detection and Correction (EDAC) and isolation, used to introduce fault tolerance features into the system.

This is a much more feasible and practical approach for micro-satellites, as can be seen by the number of missions adopting this approach in recent years.

Based on an analysis of the operations required and providing margins for higher level software modules and buffers, a 10 MHz processing system with 8 MB of SRAM and non-volatile memory was established as acceptable for the mission. The interfaces to be supported would be two CAN bus interfaces; back up serial links to Power Supply Unit (PSU) and the dual redundant Telemetry, Tracking and Commanding (TT&C) sub-systems; and two pps and serial data interfaces for dual redundant Global Positioning System (GPS) receivers. The processor chosen is the ERC32, which provides an EDAC functionality of 1-bit correction and 2-bits detection for every 32 bits of data. The devices selected are based on radiation tests results and susceptibility information available in the public domain. Cold redundant and isolatable banks of the non space-grade memory and interface are available on the system to cater for single point failures on the board. Software fault tolerance through memory scrubbing and error correction are also planned to correct single bit errors and minimise the possibility of multiple bit errors.

1.6.2 Platform Software

The platform software for the flight computer consists of the bootloader and the flight software image for the system. The bootloader is a very small and compact module that copies the operating system and application software image from the non-volatile memory to SRAM for execution and then passes control to that software. Two copies of

the image are maintained in memory. If the processor EDAC detects an error in one of the locations, it copies the corresponding location from the second image. Once the image is copied the non-volatile memory is powered off and everything executes from SRAM. This reduces the possibility of multiple bit errors developing in the data in the non-volatile memory.

The flight software is based on the VxWorks Real Time Operating System (RTOS). The features provided by the OS include concurrent process execution capability (through VxWorks tasks), file and directory services and device drivers for all interface drivers on the board. The platform software also includes a Task Manager, which is the first task spawned by the OS after booting and initialisation.

The task manager carries out self-check on power up and launches the individual application software tasks on the flight computer. Specific software processes (tasks) can be launched or terminated at any time through ground station commands. The software tasks launched by the task manager are the

i. Telemetry and status monitoring task (Spawned after initialisation)
ii. Command execution task (Spawned after initialisation)
iii. Ground interface task (Spawned after initialisation)
iv. Software agents for higher level autonomy (Spawned by ground command)

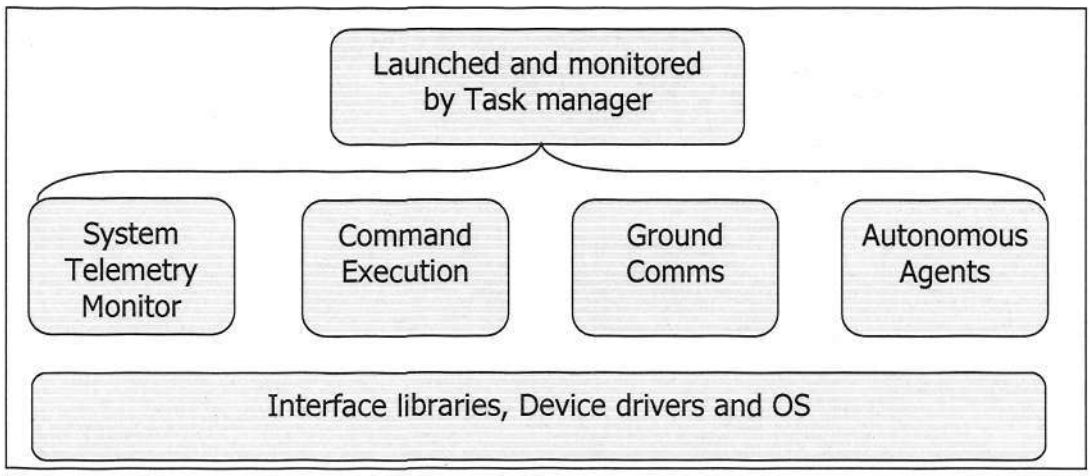

Figure 2 Tasks launched by the Task Manager

1.6.3 Baseline Operations Software

The baseline functionalities for the flight software are grouped into three sets of operations, telemetry collection, monitoring & logging; command execution and ground station communications.

The **Telemetry Monitoring** operation on X-Sat is over the CAN bus. In order to improve the determinism of the traffic on the bus, a master-slave polling scheme is used by the flight software to collect telemetry from the different sub-systems. Telemetry points required to be logged for Whole Orbit Data (WOD) information are logged at their respective frequencies. The satellite operational mode information is also maintained in the Telemetry collection module and all telemetry information is available to other modules in the flight software. The task also monitors collected telemetry and takes simple actions, such as switching off non-critical systems and going to SAFE mode, on detection of anomalies. The **Command Execution** module receives commands and executes these commands. The module supports immediate commands as well as ATCP time scheduled commands.

Ground Communications manages exchange of data between the ground station and the flight software. The commands received are sent to the command execution task and telemetry data to be sent is sent out to the TT&C sub-system. Underlying the three main processes are the libraries and APIs required for communicating with the various sub-systems on the satellite.

To support all these activities, the baseline flights software also includes API libraries and communications protocol libraries to support commanding and interactions with the other sub-systems.

1.6.4 Autonomous Mission Operations Layer

The autonomous mission operations layer would consist of autonomous agents, which can be initialised and terminated through ground commands, monitoring the satellite state through telemetry inputs and generating the commands required for achieving mission

goals. These agents would execute alongside the baseline operations software as VxWorks tasks. They would interact with the telemetry monitoring task to determine satellite state information and carry out mission operations through commands to the command execution task. Ground communication would still be carried out, when required, over the ground communications task. This provides a high level of de-coupling between the baseline flight software needed to support mission operations and the more complex autonomous agents. The software agents planned for the flight software, for the first version, consists of a Mission Planner agent, a State Maintenance agent and a Fault Handler agent.

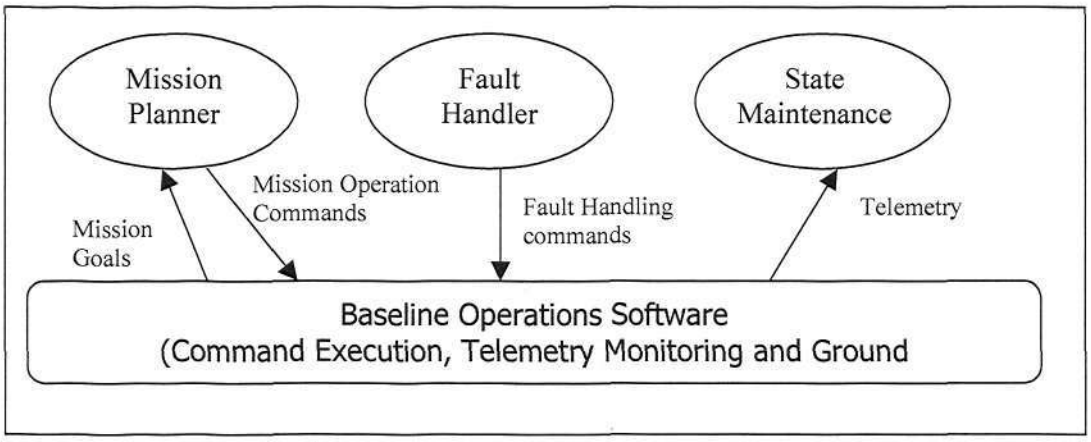

Figure 3 Autonomous Agents in Flight Computer

The mission planner receives high-level goals from the ground station and breaks them into individual commands required for the operation. Once the plan is generated, the planner monitors the state and releases each action (command) in the plan as they become ready for execution. At the end of planning, the mission planner re-samples the state information to check for validity of the plan. If the state has changed form the original initial state (excluding parameters such as time and location), the plan is cancelled and a new plan is generated.

The state maintenance agent continuously collects the received telemetry and status information and updates the state specification of the system that is maintained within the flight software. This module is also responsible to detect any anomalies in the received data and trigger fault handling operation, on detection of any faults. The fault handling

operation, once triggered, attempts to diagnose the fault, take appropriate recovery operations to prevent any further faults and try to return the system to an operational state. On detection of faults, the mission planner is inhibited. The state is maintained within the state maintenance agent. This can be sampled at any time, through request, by other software modules in the system. The planner samples the state once at the beginning of planning and once at the end, to check for consistency of state.

1.7 Organisation of Thesis

Chapter 1 gives an introduction to the field and the motivation for the project. The scope of the project, and an overview of the system being designed and its functionalities are also given. Chapter 2 gives some general background on the hardware and software aspects of the design. A study of recent missions targeting similar capability is also presented. Following this, the top-level architecture of the flight computer being designed and the breakdown of functionalities between layers is also presented.

The remainder of the thesis consists of three parts. Part I covers the hardware platform, Part II covers the baseline flight software and Part III covers autonomous operation. Part I is captured in Chapter 3, which presents the fault tolerant computing hardware developed for the computer and its reliability features. Part II is covered in Chapter 4, where the software platform and the baseline software modules developed for the flight software are described. These were developed and evaluated based on the specifications for the X-Sat mission. Part III is discussed in chapters 5 and 6. Chapter 5 sets a paradigm for high-level autonomy for micro-satellite operations and the techniques that can be used to achieve this Chapter 6 describes the design of the autonomous operations layer of the flight software.

Chapter 7 concludes this thesis with the results of the work done, current status of the system and suggests some areas where future work can be carried out.

2. Background and Approaches

2.1 Overview

This chapter gives a background on the various approaches for the design and development of an autonomous flight computer for space missions. Both the hardware perspective and the software perspective are discussed. A study of some recent missions targeting on-board autonomy are also presented in this chapter.

2.2 Hardware perspective

2.2.1 Space Environment and Electronics

The nature of the space environment, because of the extreme temperatures, radiation as well as vacuum, affects the performance of normal microelectronics devices when used in ground. The primary cause of hardware errors on space borne computing units is the radiation environment.

The other environmental sources of error are mechanical vibration and thermal cycling & extremes. However these factors are less difficult to overcome. The mechanical stress is generally only for the short duration of the launch; and the thermal range experienced on the inside of the satellite is within the tolerance limits of most electronics components. Techniques for providing higher tolerance toward these factors are available and not very costly. Besides, ground based testing of the system for thermal and mechanical qualification is often feasible and cost efficient.

The effect of radiation environment on electronics can be broadly classified into two types, Total Ionizing Dose (TID) related failures and Single Event Effect (SEE) related failures. TID is a measure of the total radiation dose or energy from radiation that is absorbed per unit mass of a device. TID is measured in Rads (1Rad corresponds to around 0.01 J/kg). TID related effects are due to cumulative long-term degradation of a device when exposed to radiation. In semiconductors devices, the ionization due to radiation (primarily protons and electrons in the LEO orbit) affects the characteristics of the substrate of the devices. The final effect depends on bias conditions and device technology; some typical effects are threshold shift in transistors and gradual increase in

the leakage currents. The total ionising dose expected radiation dosage in the planned orbit can be roughly determined and appropriate protection (such as radiation shielding using protective coating for the devices) can be employed.

A Single Event Effect (SEE) is a phenomenon that results from a single, energetic particle's interaction with the semiconductor device. SEE related problems are normally one of two kinds. Single Event Upsets (SEU) or Single Event Latcup (SEL). SEUs are radiation-induced errors in microelectronic circuits caused when charged particles lose energy by ionizing the medium through which they pass, leaving behind a wake of electron-hole pairs. This would result in observable errors such as bit flips or transient pulses in the device. SEUs are transient errors and can be normally corrected by resetting the device to its initial state. An SEL is a radiation event induced high current state that results in the device not functioning properly. Power cycling the device helps to reset the latchup. If it is not detected and handled in time, it may lead to a Single Event Burnout, where the high current caused due to the latchup results in the device being burned out. SEEs are very difficult to predict and a major source of concern for space-based systems.

Radiation environment of space is difficult to accurately simulate. Even when feasible, radiation testing is very expensive. Certain components are specially fabricated and qualified in this manner specifically for use in space. These 'Radiation Hardened' or 'Radiation Tolerant' components are usually very expensive. Their cost is the order of 10s to 100s of times the price of normal components and this is often not a feasible approach for micro satellite missions. There is therefore, an active effort in this field to use and qualify (through successful usage) commercially available components for space use. The nature of the radiation environment in the low earth orbit, combined with careful selection of components and a robust design, makes this possible.

2.2.2 Design Approaches for Reliability

Redundancy and Isolation

Redundancy consists of having multiple components or modules having the same functionality in the same design such that they can take over each other's role when

required. Thus when one of the modules or components has a problem, the backup unit can replace it. The redundant modules must be isolated from each other, in terms of power and signals, so that a fault in one may not incapacitate the other.

Redundancy can again be classified as hot or cold redundancy. If the two modules are running simultaneously it is called a hot redundancy. In the case of cold redundancy, the redundant module is in an 'off' or non-active state during the operation of the primary module, thus acting as a back up module that can be switched on when the primary module fails. An example of hot redundancy is Triple Modular Redundancy (TMR). Three independent units are used to perform the same task or measurement. An arbitrating voter unit then attempts to collect their outputs and choose the appropriate output using majority voting.

Error Detection and Correction (EDAC)

EDAC is usually used to protect data stored in the computer from bit flips caused through SEUs. EDAC involves employing methods to detect errors in the stored data and to correct them. This is done in many ways, all of them involving some form of coding. The simplest method is using some added parity bits. Commonly used codes like the Hamming code can perform Single Error Correction and Double Error Detection. Thus, an actual error occurs only when a location being accessed has two bits that have been flipped.

Scrubbing

Scrubbing is a technique that is commonly used on memory protected by EDAC. EDACs usually correct single bit errors, so an error occurs only if 2 bits on a location are corrupted. Scrubbing involves reading through and writing back the different memory locations. If a single bit error has occurred at the location, the EDAC corrects it and the location is corrected in the write back. If done frequently enough, this minimises the possibility of 2-bit errors in a single location.

2.3 Software perspective

The autonomous software modules will need to use various AI approaches and techniques in order to function efficiently. This section gives a brief overview of this field.

2.3.1 Roadmap of AI Development

The interdisciplinary field known as cognitive science grew up with the development of the modern digital computer. Researchers from such diverse fields as computer science, neurophysiology, psychology, linguistics, and philosophy were brought together by the conviction that the digital computer is the best model of cognition in general, and consequently of human cognition in particular. This forms the foundation for classical AI. This view however is now rivalled by modern approaches such as connectionism and embodiment.

Classical AI

The key feature of the classical AI approach is its treatment of the cognitive process as a syntactic activity. Classicism maintains that the rules that determine cognitive processes in cognitive systems also have two guises. On one hand they are interpretable as psychological laws governing transitions among mental states (reasoning). But on the other hand they are purely formal rules, applying directly to the syntactic structure of symbolic representations [6].

Thus, once the rules and representations of the task domain have been made precise and explicit, using some formal system of symbols and operations, these rules can be implemented and executed on conventional computing devices.

Knowledge Representation and Search

Thus, according to the classical AI viewpoint, a reasoning system must be able to abstract information from the real world into some internal representation; and have defined rules to operate on it; in order to reason intelligently. To enable this, the system must have within it the knowledge base to work and the guidelines to be used for reasoning with this

knowledge. This aspect of a reasoning system is captured under the broad field of Knowledge Representation (KR).

Knowledge representation refers to the general topic of how information can be appropriately encoded and utilized in computational models of cognition. The function of a KR is to capture the essential features of a problem domain and make this information available to the problem solving procedure. This provides a method for abstraction of data not required for the computation. The features desired of the representation method used are:

i. It must provide a natural method of representing the knowledge,
ii. It must support efficient execution of the resulting code, and
iii. It must be adequate to capture all the required knowledge.

Key Features of Classical AI

Some of the key assumptions and features of classical AI are

- Intelligent reasoning does not depend on the implementation platform or machinery, the important thing is the algorithm.
- A physical symbol system is the necessary and sufficient requirement for general intelligent action [8].
- The system is designed in a goal based top down manner.
- The system is built on a centralised information processing architecture.
- System is deliberative and deterministic
- The system can be easily implemented on any typical computing such as a computer or a microprocessor.

Drawbacks of Classical AI

Some of the main drawbacks faced with classical AI are

- Discrepancies between the virtual world formed using the symbols & rules and the real world.
- Inability of the system to perform in a novel situation in the real sense
- Real time processing limitation of centralised processing architecture

In spite of its drawbacks, a classical approach would be better suited for mission critical control software, such as the flight software. The reasons are its determinism and the limited nature of the environment.

Modern Approaches to AI

In the last couple of decades, various alternatives to the symbolic viewpoint advocated by classical AI, have found supporters and developed. Some of the major schools are the connectionist approach and that of embodiment. A brief overview of these has been included here for completeness.

A concept often seen in the modern approaches is the idea that high level intelligent action can be achieved through a number of loosely coupled peripheral processes running simultaneously. There is no central processing element. The system is thus designed in a bottom up manner focusing on the individual processes before the final goal.

Connectionist View

The connectionist approach attempts to model artificial reasoning systems based on the human brain. The human brain consists of a large number of special cells called neurons. In a simplistic sense, each neuron has multiple input signals and one output signal. Besides this, there is an activation function within the neuron, which controls the value of the output based on the input signals. There are roughly 10^{11} neurons in the brain and each neuron is connected to around 10^4 others. It is thought that this interconnected system of neurons and their respective activation functions (referred to as a neural network) enables the human brain to learn, think and reason. Artificial intelligence systems based on this architecture are called Artificial Neural Networks (ANN).

The roots of the connectionist viewpoint can be traced back to the development of the perceptron in 1950s [9]. Further work in this field slowed down in the late 1960s [10]. Renewed interest in perceptrons and ANNs started in the 1980s and significant development has been done in this area.

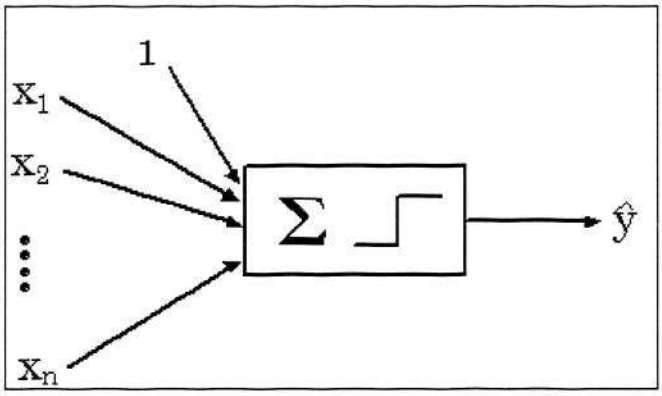

Figure 4 A Simple Perceptron

The perceptron is a simple type of ANN. The building block of a perceptron can be likened to a simplified neuron. This in itself is a preceptron. It has multiple input signals 1 bias input signal and 1 output signal. Usually, the input values are boolean, but they can be any real number. The output of the perceptron, however, is always boolean. When the output is active, the perceptron is said to be firing.

All of the inputs (including the bias) have weights attached to the input line that modify the input value. The weight is just multiplied with the input, so if the input value was 4 and the weight was -2, the weighted input value would be -8. Each unit has its respective threshold. The threshold is one of the key components of the perceptron. It determines, based on the inputs, whether the perceptron fires or not. The perceptron takes all of the weighted input values and adds them together. If the sum is above or equal to the threshold then the perceptron fires. Otherwise, the perceptron does not. These can be used as building blocks for more complex networks. Appropriate weights, connections and thresholds can bring about desired functionalities. Such a system also offers a natural scheme for learning, by modifying the weights in a network [11].

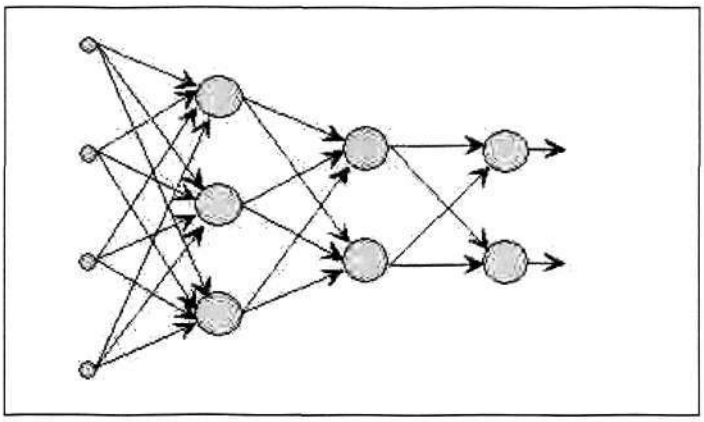

Figure 5 Multi layered Perceptron from simple units

Embodiment View

One of the fundamental assumptions of the embodied cognitive science approach is that intelligence can emerge only from real embodied physical agents interacting with their environment. Intelligence is thus an emergent quality that results from the constant interaction of the agent with its environment [12]. An important characteristic of embodied agents is situatedness.

Most of the initial growth in this area was due to its founder Rodney Brooks [13]. The focus here is not a goal based top down design, but on the interactions with the environment and how this results in emergent behaviours. Embodied agents have other features such as self-sufficiency and adaptivity, which are seen in the real world. Several architectures to implement embodied agents have been developed and are being researched in various labs around the world.

The Subsumption Architecture of Rodney Brooks is a layered approach to high level artificial intelligence [14]. Each layer gives the system a set of pre-wired behaviours. The higher levels build upon the lower levels to create more complex behaviours. The behaviour of the system as a whole is the result of many interacting simple behaviours. The layers operate asynchronously carrying out their respective functionalities.

2.3.2 Agents

An agent is defined as anything that perceives its environment through sensors and acts on the environment through actuators [7]. In the context of a software agent, the sensors are the modules that receive information from its environment and actuators are the modules that send out commands or responses to the environment. An autonomous agent is a system situated within an environment, that senses that environment and acts on it, in pursuit of its own agenda and so as to effect what it senses in the future.

Thus an autonomous agent is situated within some environment, the real world, or an artificial environment within a computing system, or within an operating system, a database, or a network. The agent actively senses it's environment, and acts upon it so as to effect what it may sense next. This interactive sensing and acting aims to achieve goals that can be expected to satisfy drives. Environments can be can be varied, some are dynamical systems some are static. The agent itself is part of the environment and some of the features required of the agent arise from the environment.

This definition of an autonomous agent requires that it pursue its own agenda. Every autonomous agent must be provided with built-in (or evolved-in) sources of motivation for its actions. These determine the operations and functionality of the agent. Functionally these systems may be serving very different purposes, and internally the approaches used by each may be based on knowledge representation schemes, neural networks or techniques from the embodied approach. The key abstraction used is that of an agent. Figure 7 gives a categorisation and breakdown of autonomous agents.

From the software perspective, software tasks are routines that are 'spawned' by the operating system or parent task to carry out some specific activity. The term software agent used in the framework of this thesis refers to those software tasks that carry out their operation in a dynamic environment with minimal human intervention and using decision-making logic to choose between available options.

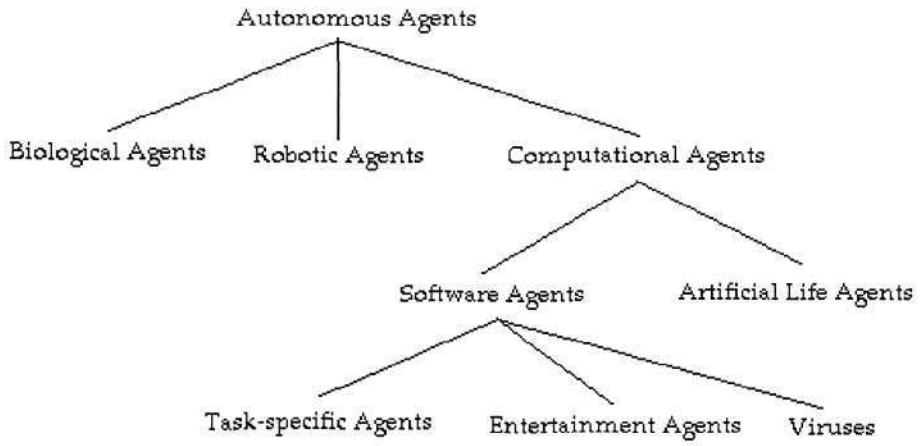

Figure 6 Taxonomy of autonomous agents

In the context of the autonomous flight software, the category of interest is that of task-specific software agents.

Multi-Agent Systems

Appropriately designed agents can handle problems of varying levels of complexity. However, complex problems are often better represented once they are partitioned into distinct parts. This also makes it easier to design solutions targeting each of the individual tasks one at a time, which makes the task easier. Often in design of agent-based systems,

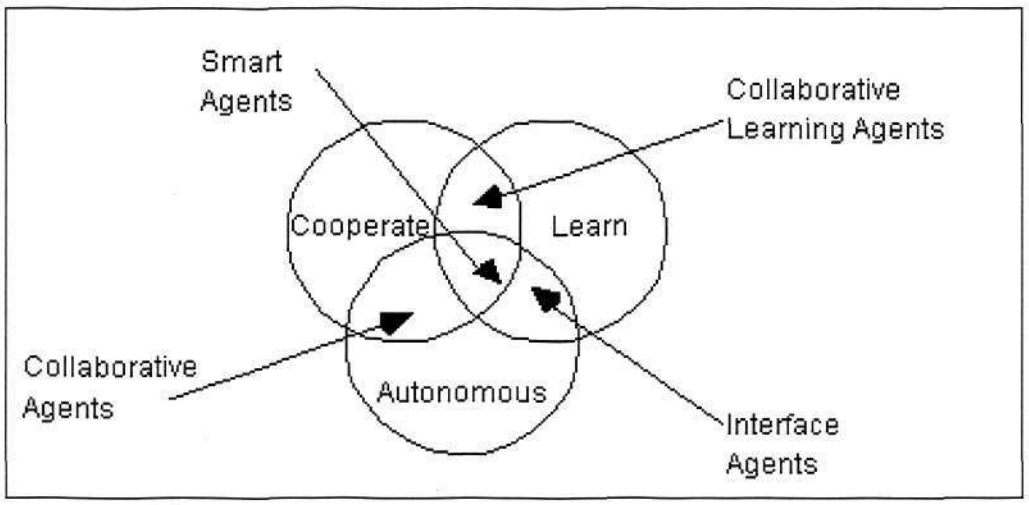

Figure 7 Collaborative Agents

complex real world problems are often divided into sub tasks and distributed among multiple agents. Such solutions are referred to as multi-agents systems. Multiple agents collaborate and act in parallel focusing on their individual tasks to produce the net required functionality of the system.

As shown in the Figure 8, collaborative agents emphasize autonomy and cooperation (with other agents) in order to perform tasks for their owners. They may learn, but this aspect is not typically a major emphasis of their operation. In order to have a coordinated set up of collaborative agents, they may have to *negotiate* or interact get information for their operations. Important issues to be handled for multi-agent systems include timing issues for communication and interactions between agents.

2.4 Autonomous Space Missions

This section briefly describes recent space missions focusing on on-board autonomy.

2.4.1 DeepSpace-1

NASA's DeepSpace-1 was among the earliest autonomous missions actually flown. The on-board autonomy for DS-1 was provided by the New Millennium Remote Agent (NMRA) architecture for spacecraft control. This architecture integrates traditional real-time monitoring and control with constraint-based planning and scheduling, robust multi-threaded execution, and model-based diagnosis and reconfiguration.

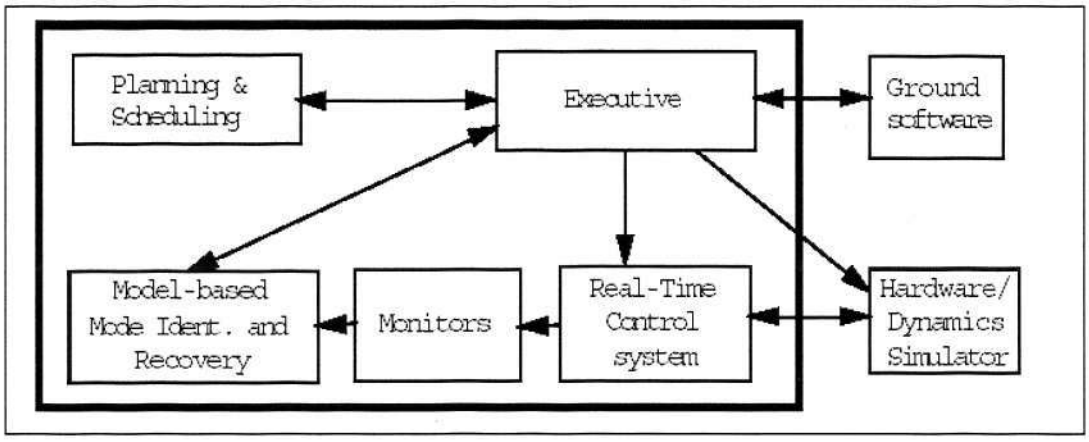

Figure 8 DS-1 Remote Agent Architecture

In the architecture autonomous operations is achieved through the cooperation of 5 distinct components shown in Figure 8. The operations on DS-1 follow the steps outlined below.

1. Retrieve high level goals from the mission's goals database. In the actual mission, goals can be known at the beginning of the mission, put into the database by communication from ground mission control or can originate from the operations of spacecraft subsystems.
2. Ask the planner/scheduler to generate a schedule. The planner receives the goals, the scheduling horizon and an initial state. The resulting schedule is represented as a set of tokens placed on various state variable time lines, with temporal constraints between tokens.
3. Send the new schedule generated by the planner to the executive. The executive will continue executing its current schedule and start executing the new schedule when the clock reaches the beginning of the new scheduling horizon. The executive translates the abstract tokens contained in the schedule into a sequence of lower level spacecraft commands that correctly implement the tokens and the constraints between tokens. It then executes these commands, making sure that the commands succeed and either retries failed commands or generates an alternate low level command sequence that achieves the token. Hard command execution failures may require the modification of the schedule in which case the executive will coordinate the actions needed to keep the spacecraft in a Safe state and request the generation of a new schedule from the planner.
4. Start again step 1 if
    a. Execution (real) time has reached the end of the scheduling horizon minus the estimated time needed for the planner to generate a schedule for the following scheduling horizon; or
    b. The executive has requested a new schedule as a result of a hard failure.

The planner/scheduler is activated as a batch process every time a new schedule is needed, and dies after a new schedule has been generated.

Schedule execution is achieved through the cooperation of the executive, a Model based Mode Identification and Reconfiguration system (MI-R), and a lower-layer of software responsible for real-time monitoring and control. The executive reasons about spacecraft state in terms of a set of component modes. The mode identi cation (MI) component is responsible for providing this level of abstraction to the executive.

MI takes as input the executive command sequence and observations from sensors to identify the current mode (nominal or failed) of each spacecraft component. The monitoring layer takes the raw sensor data stream, and discretizes it to the abstract level required by MI. Finally, the control and real-time system layer takes commands from the executive and provides the actual control of the low level state of the spacecraft.

2.4.2 PROBA

In the framework of demonstrating the feasibility of small and low-cost missions by increasing operational autonomy, ESA undertook the design, development, and operations of an actual mission dedicated to autonomy called **PROBA** (**PR**oject for **O**n-**B**oard **A**utonomy). Proba was intended to demonstrate the benefits of autonomy and to validate the supporting advanced technologies in orbit. In particular, the following autonomy functions are implemented:

- commanding for management of on-board resource and house-keeping functions;
- scheduling, preparation and execution of scientific observations (for instance: slew, attitude pointing, instrument settings);
- scientific data collection, storage, processing, and distribution;
- data communications management between Proba, the scientific users, and the ground station;
- performance evaluation and estimation of drifts, trends;
- failure detection, reconfigurations and software exchanges.

Autonomous mission planning on PROBA mission was achieved through CASPER, Continuous Activity Planning, Scheduling, and Replanning. Casper attains a higher level of responsiveness than typical planning systems through a technique called iterative

repair. When an anomaly occurs, Casper focuses on the anomaly's effect with respect to its current plan. Rather than recreating a whole new plan from scratch, Casper tries to address only the problems the anomaly introduced into the current plan. This repair operation occurs repeatedly until a valid plan emerges. Providing fast replanning enables interleaved planning and execution.

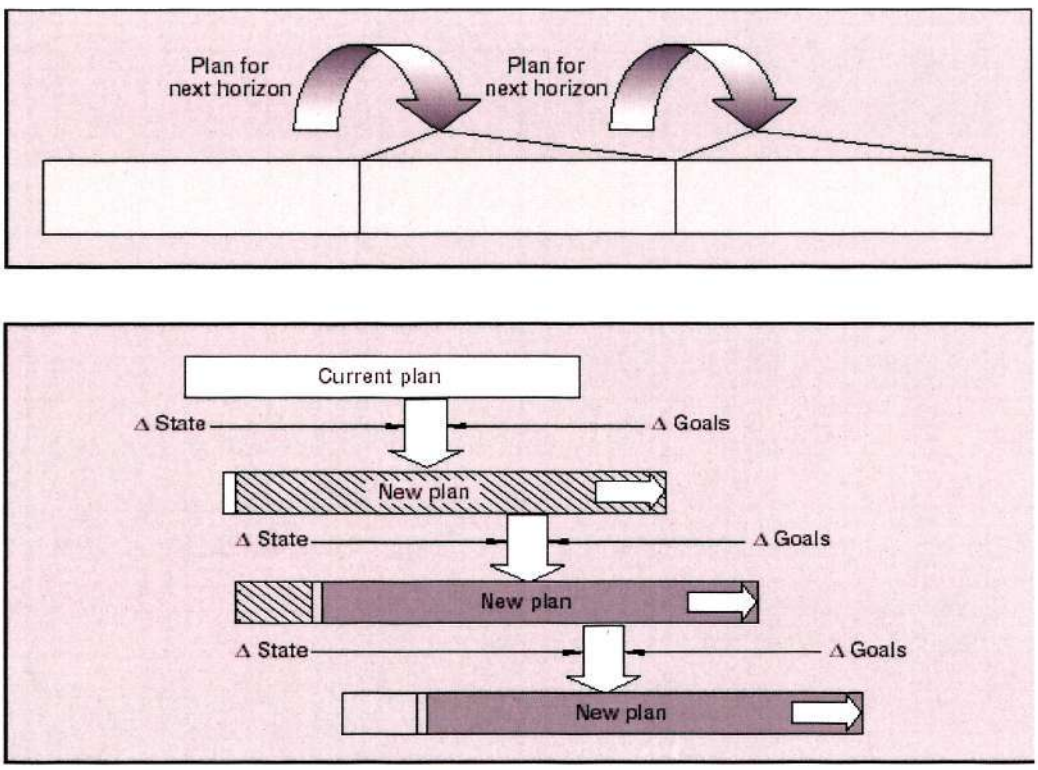

Figure 9 Typical planning vs Casper's iterative planning approach

Casper receives updates from sensors and other onboard software systems, then acts on them to ensure that the current plan is valid. All the while, this plan decomposes into commands for the subsystems aboard the spacecraft at appropriate times. [23]

Basically, Casper receives high-level goals and generates a plan. It then incrementally dispatches this plan for execution and monitors its progress. Casper stands ready to fix any plan that turns out to be flawed.

The PROBA flight computer ran under the VxWorks RTOS on ESA's space grade SPARC v7 processor, ERC-32.

2.4.3 TechSat 21 – Autonomous Science Experiment (ASE)

The Air Force Research Laboratory (AFRL) has initiated the TechSat 21 program to serve as a demonstration mission for a new paradigm for space missions. The ASE autonomous control software will fly on board the TechSat 21. This paradigm seeks to reduce costs and increase system robustness and maintainability by using onboard autonomy to enable faster response times and improve operations efficiency.

Each satellite will have 256 Kbytes of EEPROM for boot loads and 128 Mbytes of SDRAM. Communications will be through a Compact PCI bus.

The ASE onboard flight software includes several autonomy software components:

- Onboard science algorithms that will analyze the image data to detect trigger conditions such as science events, "interesting" features, and changes relative to previous observations

- Robust execution management software using the Spacecraft Command Language (SCL) package to enable event-driven processing and low-level autonomy

- The Continuous Activity Planning, Scheduling, and Replanning (CASPER) planner that will replan activities, including downlink, based on science observations in the previous orbit cycles

- Observation Planning (OP) software will enable the satellites to predict overflights of targets to facilitate on board retasking

Software Architecture and Operation

The TechSat 21 flight software is structured in a layered manner as shown in Figure 10.

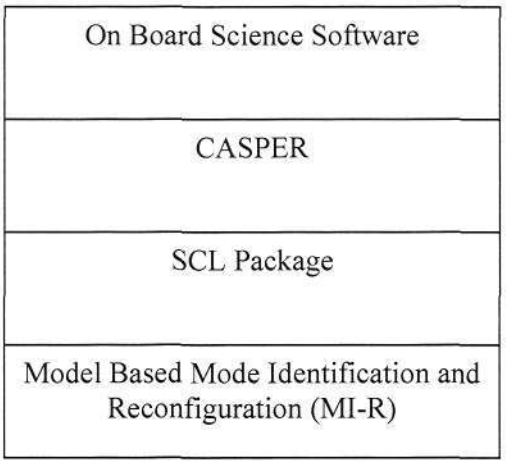

Figure 10 TechSat-21 Flight Software Layers [16]

The ASE concept of operation would be applied as follow. Initially, ASE has a list of science targets to monitor. As part of normal operations, CASPER generates a plan to monitor the targets on this list by periodically imaging them with the radar. During such a plan, a spacecraft images a river area with its radar. Onboard, a reflectivity image is formed. The Onboard Science Software compares the new image with previous image and detects that the water region has changed due to flooding. Based on this change the science software generates a goal to acquire a new high-resolution image of an area of flooding. The addition of this goal to the current goal set triggers CASPER to modify the current operations plan to include numerous new activities in order to enable the new science observation. During this process CASPER interacts with the Observation Planner to compute when the spacecraft will overfly the target and determine the required slews to acquire the target. SCL executes this plan in conjunction with several autonomy elements. Mode Identification assists by continuously providing an up to date picture of system state. Reconfiguration achieves configurations requested by SCL. Based on the science priority, imagery of identified "new flood" areas are downlinked. This science priority could have been determined at the original event detection or based on subsequent onboard science analysis of the new image. [16]

TechSat 21 is scheduled for launch in January 2006 and will fly three satellites in a near circular orbit at an altitude of approximately 550 Km. The primary mission is one-year in

length with the possibility for an extended mission of one or more additional years. During the mission lifetime the cluster of satellites will fly in various configurations with relative separation distances of approximately 100 meters to 5 Km.

2.5 Layered Architecture

Any autonomous agent or system intended for operation in a complex environment has several functionalities that it needs to exercise. The functional capability to support these must be built into the agent. These capabilities can be organised into functional layers in their criticality or level of complexity.

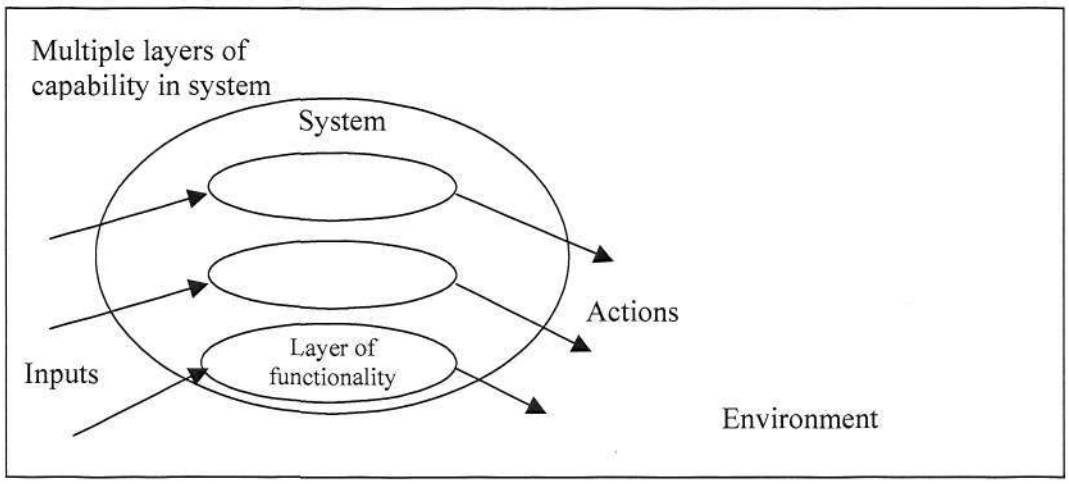

Figure 11 Multi-layered system in operating environment

Following this approach, the system can be designed in a layered manner where each layer represents certain clearly defined functional capabilities targeting certain activities. The layers interact with and make use of functionality and data provided by the layers below them. Pre-defined interfaces provide the support for the required interaction between the modules. This approach is loosely based on the Subsumption architecture proposed by Rodney Brooks.

A functional layer has clearly defined functionalities and interfaces required of it. Each layer makes use of the functionalities and data provided by the layers below it and provide these additional functionalities. The system is thus built in a bottom up manner with the lowest layer providing the lowest level of operation and each additional layer

adding incremental higher-level functionalities. The internal design used by a module to achieve its functionality is not tightly coupled with the other modules and the rest of the system.

Since the layers add-on capabilities incrementally, a failure in one layer potentially affects the layers above it, but layers below it are not compromised. They do not depend on the higher layers' functionalities for operation. So the criticality and reliability level required is highest for the lowest layer and can decreases with each higher layer. Thus the system consists of layers of increasing complexity and functionality as we move up the layers, and criticality and reliability requirements increasing as we move down the layers.

2.5.1 Key Features

The key features of breaking down the system in a layered fashion as described above are:

Modularity and reduced complexity: Instead of treating the system and its requirements as a whole, this approach enables breaking up of the requirements into modules. The net complexity of the system is thus broken down by dividing up the requirements among the capability layers, with each layer having to concern only with its own requirements. Issues related to coupling of requirements and prioritisations of operations are handled through the layer priorities and interface capabilities.

Scalability: Designing the system in this manner enables development of the system in a layer by layer fashion. The basic layers can be built and tested, and further layers can be added on step by step. The system can be scaled down by removing capability layers that are not required or by modifying implementation of specific layers to suite requirements.

Ease of reconfiguration and upgrade: If interfaces defined are maintained, reconfiguration of a particular layer or module can be carried out without affecting the rest of the system. This provides the support for dynamic re-configuration and upgrades where required.

Reliability and robustness: The increase in complexity among the layers and increase in reliability and criticality down the layers gives an intrinsic protection to the system in case of faults in the higher layers. The system can still be diagnosed and controlled by communicating directly to the lower layers.

PART – I (Hardware)

3. Fault Tolerant Hardware Platform

3.1 Overview

The lowest and most basic layer needed for an autonomous computer is a reliable hardware platform. Unlike general terrestrial environments, space is an extremely harsh and harmful environment for electronic systems. The platform must be designed taking into consideration this environment; and must be able to provide reliable functionality for the lifetime of the mission.

An overview of the functionalities of the flight computer hardware is listed below:

- It must provide the computing hardware (processor and memory) required for the flight software,
- It must support I/O interfaces for the CAN bus as well as the various dedicated links needed for communications

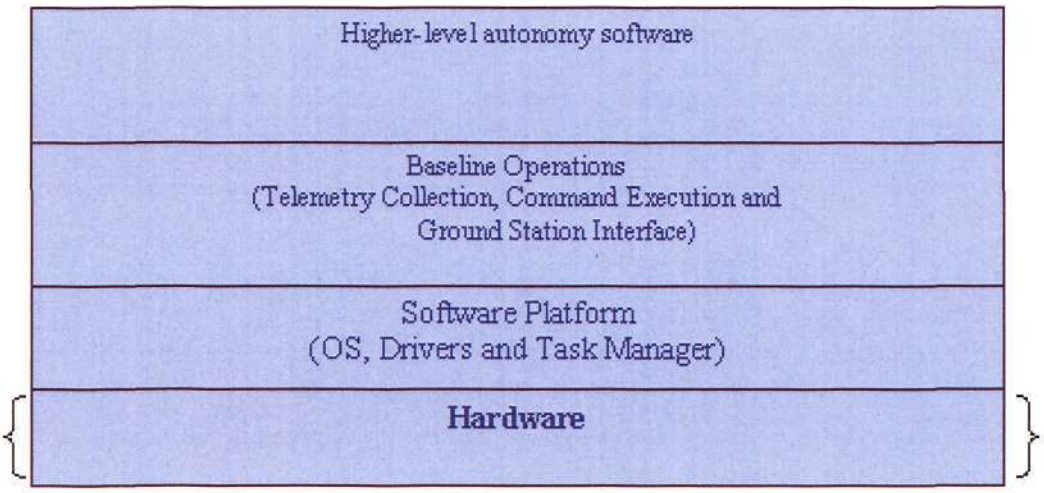

Figure 12 Hardware Platform in the context of the Flight Computer Layers

As the diagram shows, all other functionalities of the flight computer depend on a reliable hardware platform.

3.2 OBC Hardware Design

The OBC hardware is based around the space grade ERC-32 processor. Two main guidelines used in the design of the OBC hardware avoiding single points of failure and employing redundancy and isolation for non-qualified components. The block diagram in Figure 13 shows the different entities constituting the OBC hardware platform.

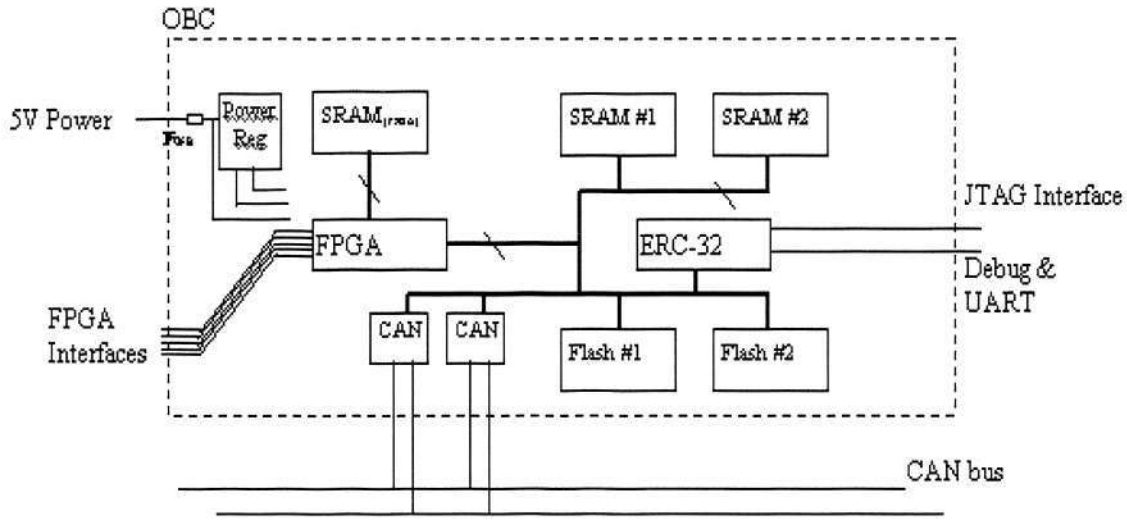

Figure 13 Block Diagram of On Board Computer Hardware

Detailed Description of Hardware

The various blocks constituting the OBC hardware are described in detail in this section.

Microprocessor

The TSC695FL (ERC-32) microprocessor from Atmel has been selected for the OBC board. This is a SPARC v7 based processor qualified for space applications. The processor is used in 32-bit mode and supports Error Detection and Correction (EDAC) of data accesses using additional check bits on the data bus. The data bus therefore consists of 32 bits of data + 8 check bits for EDAC. The EDAC offers 1-bit error correction and 2-bit error detection on every 32-bit word.

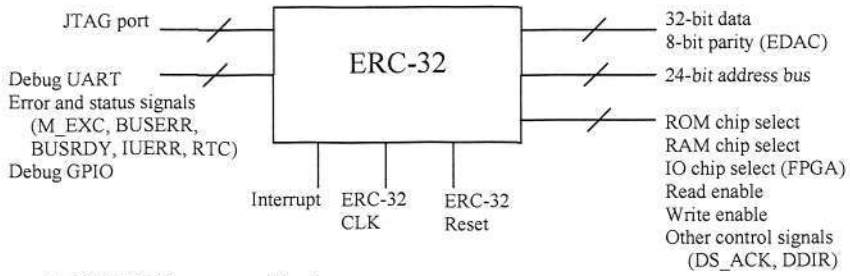

Figure 14 ERC-32 Processor Block

The SRAM and non-volatile memory access is protected by EDAC using the 32 + 8 bit bus. The FPGA interface uses an 8-bit data & 11-bit address bus [TBC] and is not EDAC protected. A JTAG interface as well as 2 UART interfaces are supported and pulled out for development and debugging purposes.

The Clock and Reset pins of the processor are pulled to the FPGA. These can be configured to be driven by the FPGA or directly from the clock and reset buttons as required. The processor is qualified for a speed of 15MHz. As is practise with several space missions, the TSC695FL is clocked at a de-rated rate of 10MHz (20MHz clock internally scaled down to 10MHz).

FPGA

The RT54SX-S radiation tolerant FPGA from Actel is the FPGA used for the OBC board.

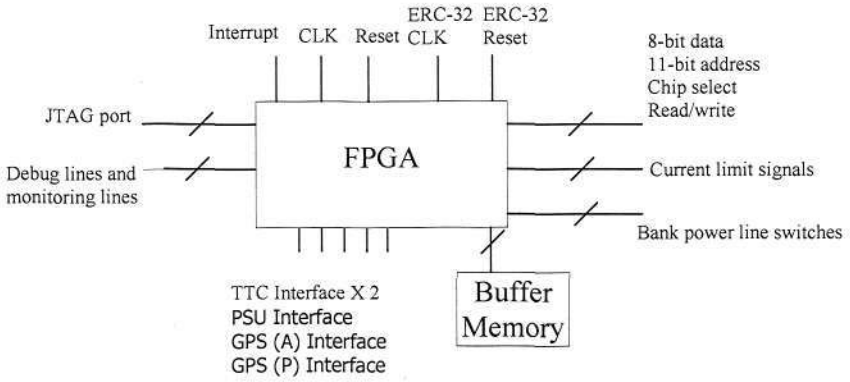

Figure 15 FPGA Block

As mentioned earlier, the FPGA supports an 8-bit data and 11-bit address bus from the processor along with the Chip Select, Read/Write and Reset signals. The FPGA also provides the 10 MHz clock and the Reset signal for the processor at boot up.

The FPGA is connected to the various interfaces that need to be supported by the OBC (excluding the UART and JTAG interfaces from the processor). The FPGA also supports the current limit sensing and bank switching operation on the OBC.

FPGA Buffer memory:

In order to support the data-buffering feature required of the FPGA, an additional dedicated SRAM memory is provided to the FPGA. This is planned to be 512Kbytes in size. The device planned is the TC55VBM416AFTN55 from Toshiba during development and the 33LV408RPFS from Maxwell for the flight version. The access bus will consist of 8 bits of data and 19 bits of address lines along with chip select and read/write control signals. The data in the buffer memory is not protected.

Clocks

The onboard clock is connected directly to the FPGA. This is a 40 MHz clock. All clocks required on the board (processor – 20MHz & CAN controllers – 16MHz) are provided by the FPGA.

SRAM memory

Two banks of SRAM memory are available on the OBC board. Each bank is 8MB consisting of commercial grade memory devices. The device selected is the TC55VBM416AFTN55 chip from Toshiba. This is an 8-bit access 2 MB chip.

The estimated size of the flight software is approximately 400KB for the OS (VxWorks) and about 1.5MB for the application layer of the flight software and necessary data structures. This comes to roughly 2MB. Using a scaling factor of 2 for memory sizing, a memory size of 4MB is desirable on the OBC for normal mode operation. An additional SRAM memory of 4MB is included to support additional software modules forming a memory bank of 8MB for normal operations.

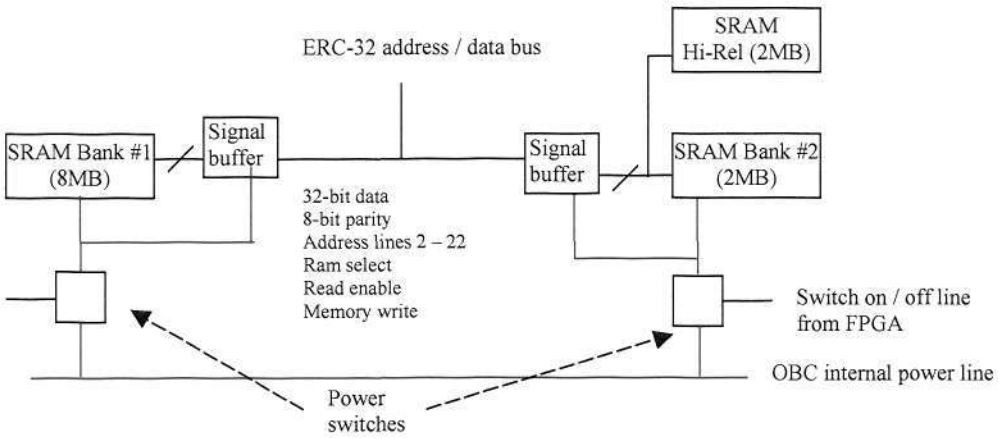

Figure 16 SRAM Banks Block

Each bank requires 5 chips to make up the 2MB (8MB of data + 2 MB of check bytes). Each 40-bit access involves 8 bits from each of the 5 devices. Thus, all accesses will be in multiples of four bytes and address lines 2-22 of the processor are connected to address pins 0-20 of each SRAM device. The address and data lines are driven through one-way and two-way buffer chips respectively.

Non-volatile memory

Two banks of non-volatile storage memory are available on the OBC board. Each bank is 8MB of commercial grade memory. The device is the TE28F016B3 from Intel [.

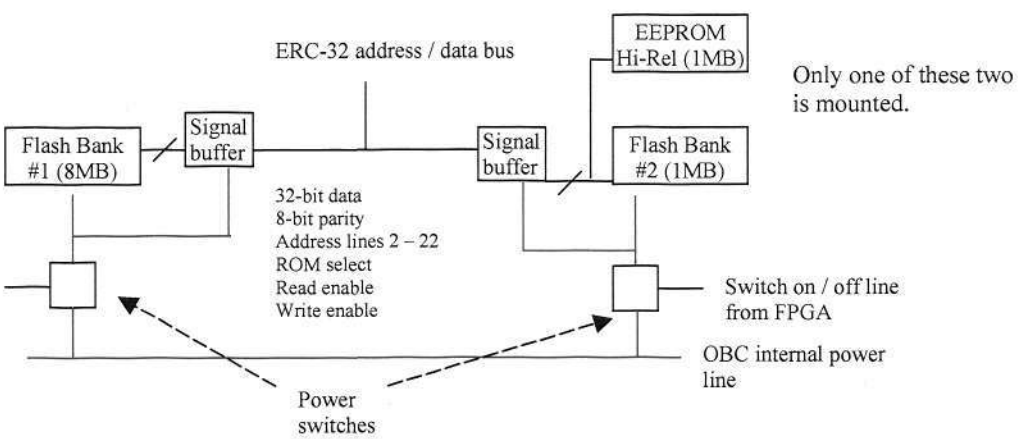

Figure 17 Flash Banks Block

As in the case of the SRAM, the bank requires 5 chips to make up the 2MB (8MB of data + 2 MB of check bytes). Each 40-bit access involves 8 bits from each of the 5 devices. Thus, all accesses will be in multiples of four bytes and address lines 2-22 of the processor are connected to address pins 0-20 of each Flash device.

Switching, and Signal / Power Isolation

The various banks on the board are designed such that they can be individually switched off in case of hard errors without impact on the rest of the electronics on the board. This is particularly for protecting the board as a whole from internal power shorts in specific banks that may be caused due to latch ups and burnouts in devices. This feature gives the OBC the capability to isolate and confine hard errors on devices.

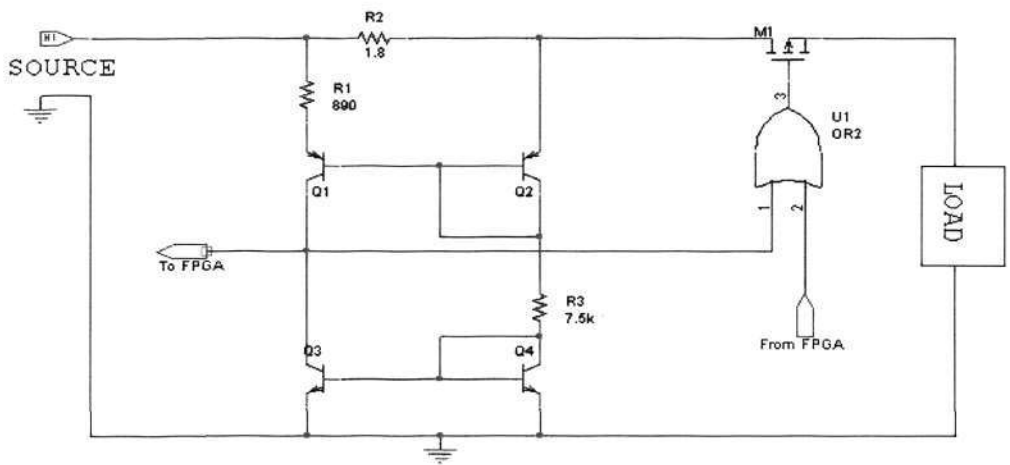

Figure 18 Current limiting circuit

The power supply line (devices' Vcc) to each bank passes through a current-limit sensing circuit, the output signal of which is monitored by the FPGA. The FPGA in turn controls a switch through which it can switch on or off the bank. All signal lines between banks and the microprocessor pass through buffer chips pass through buffer chips. This provides isolation to the bus, in case of a failure on one of the devices. The buffer devices used are 54AC244 and 54AC245 chips from National Semiconductor.

Power Supply

The OBC board takes in 5V from the power supply and provides the devices on board with 5V (direct from the PSS); 3.3V (regulated) and 2.5V (regulated) voltages. The regulators selected are the LM1117 processor (3.3V and adjustable) from National Semiconductor.

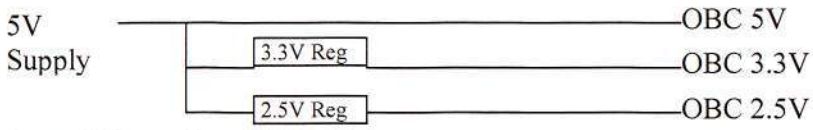

Figure 19 Power Input Block

CAN controller and CAN Bus Interface

The CAN bus link from the processor consists of the CAN controller device which implements the logical CAN protocol standards and the CAN bus transceiver that converts the TTL signals to the electrical specifications for the CAN bus. The link to the ERC-32 consists of 8-bit address and data buses along with chip select, read/write, reset and interrupt control signals. The TJA1054 CAN transceiver chip from Intel connects the OBC to the CAN bus. The controller is configured to operate at a physical speed of 80kbps over the CAN bus.

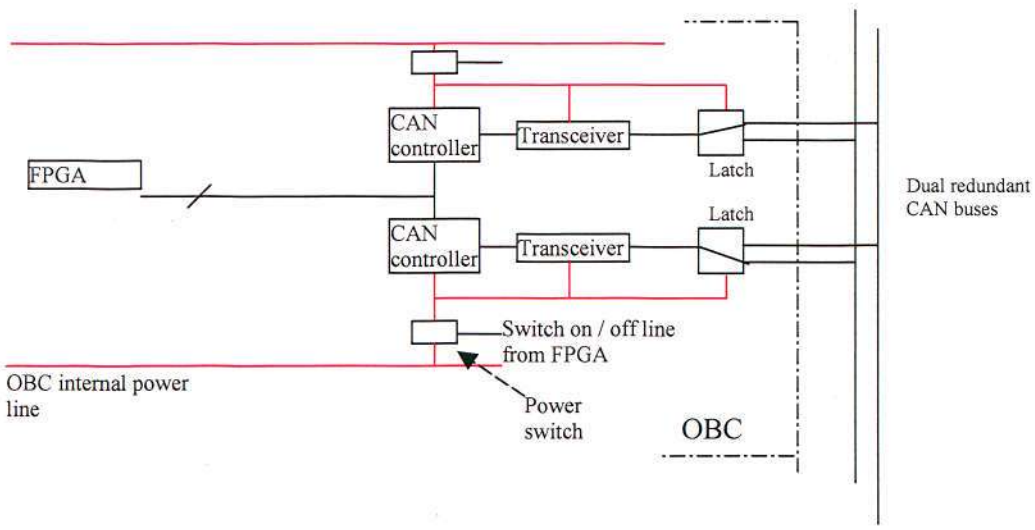

Figure 20 CAN bus Interface

Two hot redundant CAN interface banks are present on the OBC to handle single point failures and increase reliability. The CAN bus on X-Sat is also dual redundant. The output of the TJA1054 transceiver connects to the required bus through a switching latch controlled by the FPGA. The latch device is the TN-L2-5V from NAIS.

PSU Digital Interface

The dedicated interface to the PSS consists of full bi-directional UART over RS422 links. Two RS422 transceivers are provided on the OBC board dedicated for transmit and receive channels respectively. These are connected to the FPGA on the OBC. The RS422 transceiver device selected for the OBC is the LTC1480 from Linear Technology. These operate at 9.6kbps physical link speeds.

TT&C Interface

Configuration link: The configuration link to the TT&C consists of full bi-directional UART over RS422 links. Two RS422 transceivers are provided on the OBC board dedicated for transmit and receive channels respectively. These are connected to the FPGA on the OBC. The RS422 transceiver device selected for the OBC is the LTC1480 from Linear Technology. These operate at 9.6kbps physical link speeds.

Synchronous downlink: The synchronous link to the TT&C is a one-way downlink path from OBC to TT&C. The link consists of differential data, clock and frame synchronisation lines. LTC1480 transceivers connected to the FPGA are provided for these signals.

Asynchronous uplink: The HDLC-m uplink to the TT&C consists of one-way Hamming code protected UART over RS422 link. An RS422 transceiver is provided on the OBC. This is connected to the FPGA on the OBC. The FPGA handles the Hamming code error detection and correction. The RS422 transceiver device selected for the OBC is the LTC1480 from Linear Technology. These operate at 19.2kbps physical link speeds.

RamDisk Interface

Command and Configuration link: The command and configuration link to the RamDisk consists of full bi-directional UART over RS422 links. Two RS422 transceivers are provided on the OBC board dedicated for transmit and receive channels respectively. These are connected to the FPGA on the OBC. The RS422 transceiver device selected for the OBC is the LTC1480 from Linear Technology. These operate at 9.6kbps physical link speeds.

Synchronous link: The synchronous link to the RamDisk is a semi bi-directional interface between OBC and SSR. The link consists of differential data, clock and frame synchronisation lines. LTC1480 transceivers connected to the FPGA are provided for these signals.

Differential Reset: The Reset signal for the RamDisk is provided through an LTC1480 transceiver connected to the FPGA.

GPS (Accord) Interface

Serial data link: The serial data link from the GPS receiver is supported by an RS422 transceiver connected to the FPGA on the OBC. The RS422 transceiver device selected for the OBC is the LTC1480 from Linear Technology. This operates at 9.6kbps physical link speeds.

Logic Signals and Ground: The manoeuvre, reset, watchdog timer and watchdog enable signals from the GPS receiver are connected directly to the FPGA. The ground is connected to the local signal ground reference of the OBC.

1 PPS Input: The 1 PPS input from the GPS receiver is connected to an input pin on the FPGA.

GPS (Phoenix) Interface

Serial data link: The serial data link from the GPS receiver is supported by two RS422 transceivers connected to the FPGA on the OBC. This is a full-duplex link. The RS422

transceiver device selected for the OBC is the LTC1480 from Linear Technology. This operates at 19.2kbps physical link speeds.

Logic Signals and Ground: The programming-enable and reset signals from the GPS receiver are connected directly to the FPGA. The ground is connected to the local signal ground reference of the OBC.

1 PPS Input: The 1 PPS input from the GPS receiver is connected to an input pin on the FPGA.

One – PPS Output

Star Tracker: The pps output for the start tracker is pulled directly from the FPGA to the connector.

IRIS / ADAM: The pps output lines for IRIS and ADAM is shared (as they are operated mutually exclusively) and pulled directly from the FPGA to the connector. During the IRIS operation this line is expected to generate a PPS signal. During ADAM operation this line is used for the time synchronisation.

Development and Diagnosis

UART: The UART output from the processor is pulled to a MAX3232 transceiver and to the DSUB-9 debug connector.

JTAG: The JTAG interface lines from the microprocessor are also pulled to the DSUB-9 debug connector.

3.3 FPGA Functionality

The FPGA has two roles in the OBCs operation. The first role is as an interface peripheral. The interfaces connecting to the TT&C link, the PSU digital link, the RamDisk link, the two GPS receiver links and the PPS output for X-Sat sub-systems are all handled by the FPGA. The FPGA in turn is memory mapped into the ERC-32 as an

I/O peripheral, over a parallel data/address bus. The ERC-32 uses this common interface to operate and interact over all the external interfaces supported by the FPGA.

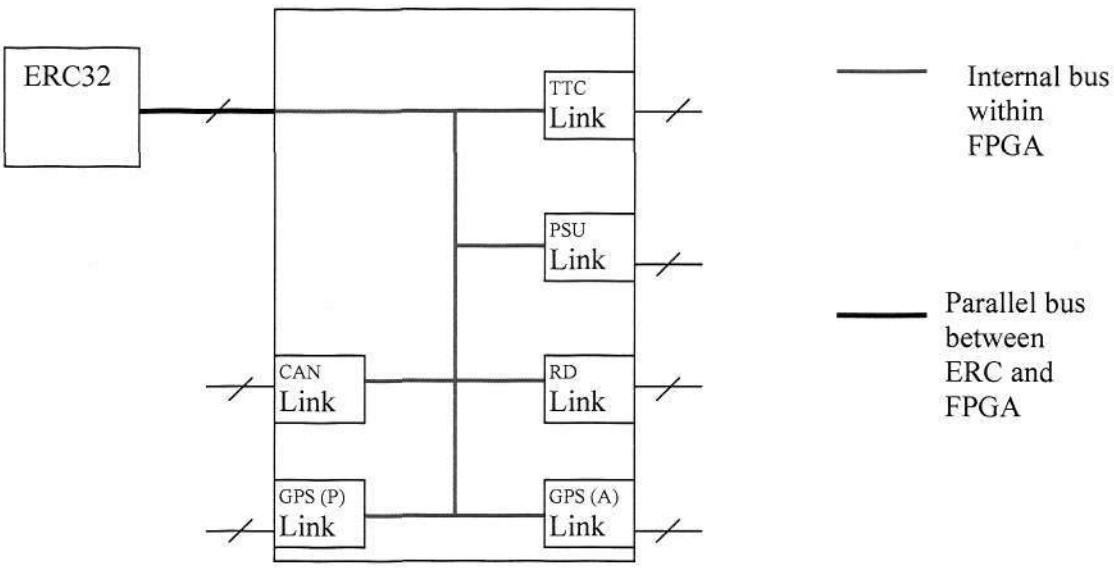

Figure 21 FPGA Communication links and ERC-32

The other role of the FPGA is to monitor current and switch between primary and redundant banks when high current is detected; both during OBC boot up and during normal operation.

PART – II (Baseline Flight Software)

4. Baseline Operations Software

4.1 Overview

This chapter describes the baseline operations software of the flight software. These provide the functionalities to command and monitor the spacecraft. The baseline software can be partitioned into two, the platform software consisting of the OS, device drivers and other services (file, time etc.); and the baseline functionality modules. The satellite mission can be controlled through the interfaces provided by this layer, both by ground commands or higher layer software.

The baseline functionalities for the OBC are grouped into three sets of operations, telemetry collection, monitoring & logging; command execution and ground station communications.

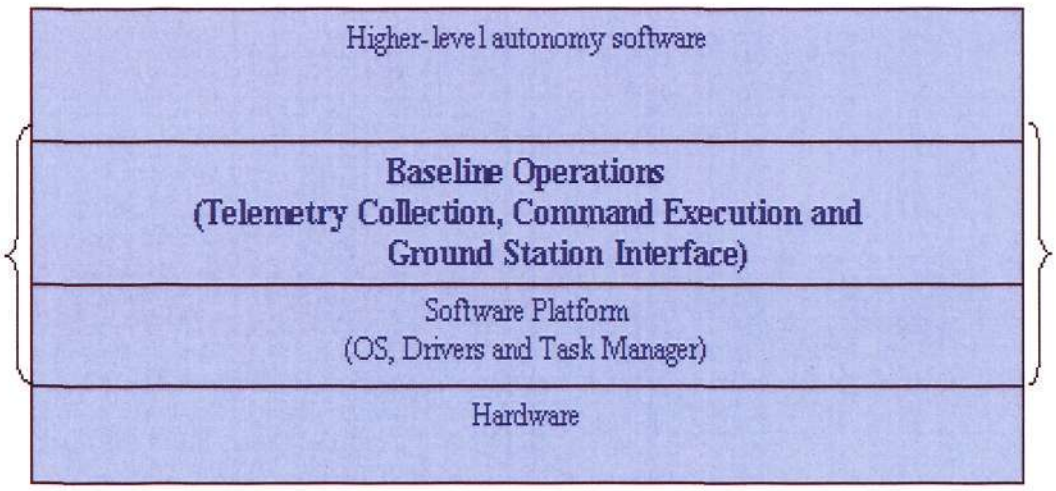

Figure 22 Baseline Flight Software in the context of the Flight Computer Layers

In order to detect activities, status and potential anomalies on the satellite, the software must continuously collect telemetry data generated on different sub-systems and log them for future diagnosis. Besides collection and logging, a certain amount of monitoring of received values must be done to detect any anomalies from the expected range of values and put the satellite to SAFE mode. Together, these form the telemetry collection, monitoring and logging activities.

In order to control the mission, the flight software must also be able to execute commands that change operational software parameters, such as telemetry collection rate etc. or send commands to other sub-systems to operate them. The command execution module provides this capability. The ground interface module provides the interface for the ground station operator to control and monitor activities on the satellite.

Besides these software processes, the baseline flight software also includes the interfaces and the libraries and API required for communication with external sub-systems. These are common and shared by all the tasks for their operation.

4.2 Software Platform

The first software unit to execute in the flight computer is the bootloader. The bootloader is a very small and compact module that copies the operating system and application software image from the non-volatile memory to SRAM for execution and then passes control to that software. Two copies of the image are maintained in memory. If the processor EDAC detects an error in one of the locations, it copies the corresponding location from the second image. Once the image is copied the non-volatile memory is powered off and everything executes from SRAM. This reduces the possibility of multiple bit errors developing in the data in the non-volatile memory.

The flight software runs on the VxWorks RTOS, which has been used extensively in space for several missions. The various processes of the flight software are grouped as VxWorks "tasks", which are similar to independent software processes. The software platform includes the boot loader, the vxWorks operating environment, drivers for all peripheral devices on the board and a task manager. The facilities that the vxWorks image provides are the drivers to interact with the FPGA; time maintenance functionality; file system services; and task spawning and inter-task communication features.

The task manager launches the individual flight software application tasks. Required software processes (tasks) can be launched or terminated at any time through ground station commands.

The software tasks launched by the task manager are the

i. Telemetry and status monitoring task
ii. Command execution task
iii. Ground interface task
iv. Software agents for higher level autonomy

Details regarding each of these processes are discussed in later chapters. The task manager also provides a software watchdog timers for each of these tasks to detect potential software hangs and re-initialise the tasks. The task manager periodically generates a heartbeat message, for an external watchdog timer, to indicate activity on the flight computer. This is for an external watchdog timer to RESET or power cycle the system in case some error causes it to hang. The task manager is thus, the highest priority task within the flight software.

4.2.1 Operation

At power up, all the non-space-grade banks of the flight computer are in switched off state. The FPGA is the first unit on the board to power up. It holds the ERC 32 in reset state as the board powers up. The FPGA then goes into a simple sequence of start up tasks. This provides a reliable entry point as the FPGA does not depend on any other logic devices for its operation. One by one it powers up the non-space grade memory banks. Each unit is powered up, held on for a short period while it checks the current monitoring circuit for any glitches in current drawn. If any unit is in error, the redundant unit is woken up and the process is repeated. Once a valid Flash bank and SRAM bank are detected, the ERC 32 is released from reset state. Thus, so long as one bank of Flash and one bank of SRAM are operational, the processing core is woken up and the software can run.

Once the ERC 32 starts, the boot-loader code in the active Flash copies the main flight software from itself into SRAM. Once this is done, control is passed to the code in the SRAM. The Flash bank is then switched off and execution continues in the SRAM. As mentioned earlier, this protects the Flash from SEE related errors that may accumulate to

multiple bit errors in the code in the Flash as error rate in Flash is reduced several fold when it is not powered.

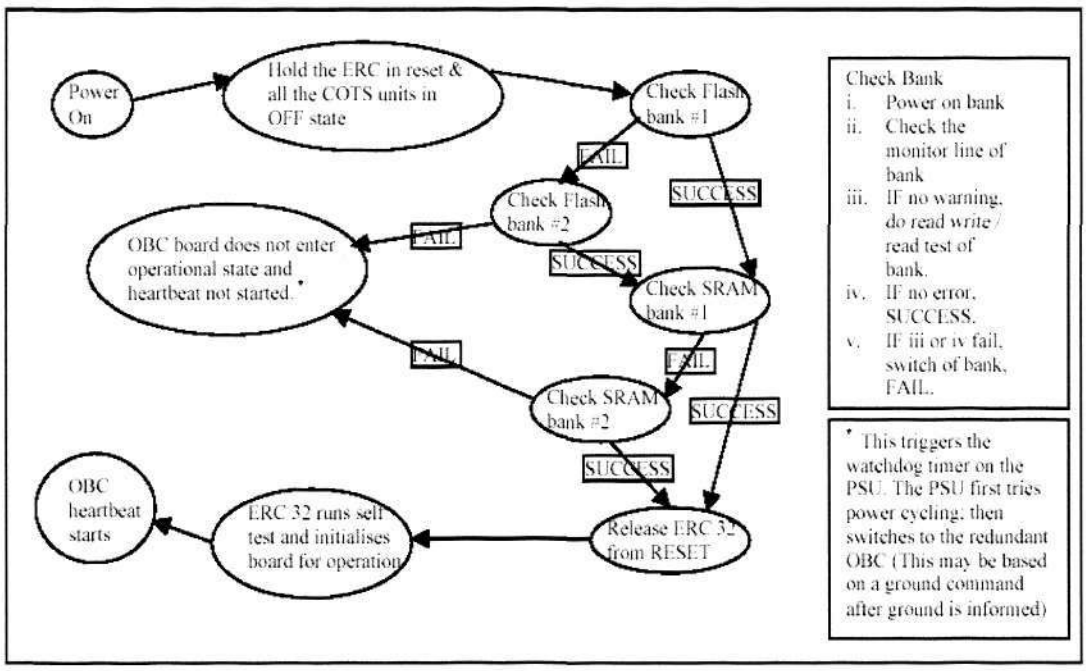

Figure 23 Power up sequence

The first process initiated, after the VxWorks boot up sequence, is the Task Manager. The Task manager does a wake-up self-test, initialises interfaces and initiates the rest of the flight software processes. If any task fails to reset its watchdog timer, it is stopped, memory cleaned and the task manager reinitiates the process. The task manager also continuously carries out memory scrubbing of the SRAM memory during normal operation. This operation corrects any single it errors in memory because of the built in EDAC in the ERC-32, which corrects single bit errors. Thus, accumulation of single bit errors and probability of double bit errors in a word are reduced. The frequency of scrubbing is modifiable through ground command. Data about error rates can be collected as telemetry and studied in the orbit-commissioning phase to fine tune the memory scrubbing frequency. The default value will be kept as a high rate of four times a day,

which takes into consideration and allows for the expected operations and loads on the system.

4.3 Telemetry Collection and Monitoring

The telemetry collection operation on X-Sat is over the CAN bus. In order to improve the determinism of the traffic on the bus, a master slave-polling scheme is used over the bus to collect telemetry from the different sub-systems. Each telemetry point is polled and sampled periodically at their respective frequency. Similarly, telemetry points required to be collected for Whole Orbit Data (WOD) information are logged at their respective frequencies. The satellite mode information is also maintained in the Telemetry collection module and the information is available to all modules within the software.

4.3.1 Operation

The telemetry collection configuration in the flight software is done through a look up table (LUT) that contains the parameters for the collection of each telemetry point. The table contains an entry corresponding to each telemetry point on the satellite. The following information about every point is maintained by the LUT. Sample Enabled; Sample Period; Sample Delay; Log Enabled; Log Period; Log Delay; RT enabled. Sample enabled and Log enabled represent whether the particular telemetry point should be sampled and logged respectively. The Sample and Log periods specify the frequency of sampling and logging of the telemetry. The Sample and Log delays specify the time to the next sampling or logging instant. All the time periods are specified in multiples of 1 second, the operation cycle of the telemetry collection operation. The RT enabled field indicates whether the particular telemetry point is to be sent down as real time telemetry or not. Telemetry points that are included as part of real time telemetry to the ground station are logged into a dedicated real time telemetry buffer, which is a cyclic buffer.

The telemetry collection operation cycle takes place once every second. Every operation cycle, the telemetry-monitoring task goes through each entry of the LUT. If the telemetry point sample enable entry is '1', the sample delay entry is decremented. If the sample delay entry is '0' a request is sent on the CAN bus for the particular sub-system and

telemetry point, and the sample delay is changed from '0' to the sample period value. The received telemetry value is stored into a buffer where it will be retained until the next time it is sampled. This buffer forms the Current telemetry data structure.

Telemetry Point	Sample Enable	Sample Period	Sample Delay	Log Enable	Log Period	Log Delay	RT Enable
$Current_{TTC}$	1	5	4	1	10	7	1
$Temp_{XBT}$	0	0	0	0	0	0	0
$Status_{PSU}$	1	1	1	0	0	0	0
....							

Table 1 Telemetry Collection LUT

When the telemetry value is received, the corresponding log enabled field is checked. If it is '1', the log delay entry is decremented. If it is '0', the value is saved to the WOD file in the memory and the log delay is changed to the log period value. The WOD file is a cyclic file. Once it is full, the writing starts again at the beginning erasing the oldest saved values.

The monitoring operation is managed through another look up table. The LUT has entries for the maximum and minimum value of each of the collected telemetry measurements. If the sampled value is outside this range, collected telemetry values are logged into a report for the ground and the satellite is put into SAFE mode. The monitoring operation can be enabled and disabled from higher layers or by the ground.

4.3.2 Interfaces

The telemetry collection module provides two kinds of interface.

i. Configuration interface: The configuration interface is used to modify entries in the LUTs that control the collection. The mode information can also be changed through this interface.
ii. Telemetry data retrieval interface: The telemetry data retrieval interface is used to retrieve collected telemetry data and mode information, from the current TM structure or from the WOD file. These interfaces are used by the ground and higher layer

software to control collection and obtain satellite telemetry information. The data can be retrieved as individual parameters or as a block.

4.4 Command Execution

The commands that the flight software receives can be classified into two types, based on their end destination. The first type is commands whose end destination is the flight software. These include commands that configure the operation parameters of the software such as telemetry collection parameters, commands to retrieve WOD information, etc. The second type is commands whose end point of execution is another sub-system. These are just to the appropriate destination for execution. These include commands to switch on and off other sub-systems, operate the payload, etc.

The commanding operation is over the CAN bus and uses a command-response scheme. The response indicates that command has been received and is used by the software as the acknowledge ('ACK') to the command.

Command execution is of two types, based on their scheduling. Immediate commands (IMCD) are commands that are executed as soon as they are received. Time tagged commands are commands that are executed when their execution time is reached. The command execution module provides command scheduling with 1 second resolution. The time tagging uses the Absolute Time Command Processor (ATCP) scheme where the time is specified in absolute time in the command.

4.4.1 Operation

The command execution module maintains two command queues for its operation. The immediate commands queue and the ATCP scheduled commands queue. Twice every second these queues are checked for commands to be executed. Any commands in the immediate commands queue, and ATCP commands whose tagged time matches the current time are executed.

The command execution module is organised into the command queue parser and the various execution libraries. The libraries contain the execution operation for each of the commands. They are organised by sub-system. The command queue parser first looks through the ATCP commands queue, and then the immediate commands queue.

The commands are grouped in to the following suites depending on their functionality:

i. Flight computer Configuration
ii. ADCS commands
iii. TT&C commands
iv. PSU commands
v. TM and status retrieval
vi. Ground Interface commands
vii. X-Band Operation commands
viii. IRIS Operation commands
ix. ADAM Operation commands
x. PPU Operation commands

Each command suite library contains a second level parser to handle extraction and execution of the commands in the particular command suite. The command queue parser identifies commands that are ready and the command suite that it belongs to. It then passes the command to the second level parser of the command suite library. The second level parser identifies the specific command number, extracts the command arguments, and executes the appropriate command. The command is then removed from the command queue.

Once a command to an external sub-system is sent, the command execution modules expects the corresponding acknowledge message within a given timeout period. If the acknowledge is not received, the command is re-sent. This is done 3 times. After 3 attempts, the command is aborted and logged into a report, which can be used by the ground for diagnosis. All received responses are logged in a cyclic buffer and can be retrieved by higher layers or the ground.

4.4.2 Safety Layer

The ground station uses time tagged ATCP commands for advance scheduling of operations that are to be carried out when the satellite is out of ground station contact. These commands go into the ATCP commands queue and are executed at their tagged time. However, unexpected or unforeseeable situations make it undesirable to execute the command at the originally scheduled time. For example, the ground station may schedule an operation of the ADAM payload in the next orbit. After the commands are received and awaiting to be executed, some fault causes a drop in the battery power available for the satellite. Executing the scheduled ADAM commands would cause an even further drop in the battery and potentially permanent damage to the power supply. Therefore it is necessary to have a built in mechanism to avoid such situations, even after the commands are scheduled.

For this, the command execution module has a 'Safety Layer' that validates all commands that are ready, prior to execution. The safety layer maintains a database that has a list of constraints that have to be satisfied, before the command can be executed. If any of the constraints are not met, the command is aborted and the command, as well as the reason for it's failing the safety check, are logged into a report for the ground. The entries in the safety layer database can be modified through ground commands.

4.4.3 Interfaces

The command execution module provides interfaces for the following functionalities to higher layers and the ground.

i. Add or Remove commands into the ATCP / Immediate commands queue
ii. Modify entry in safety layer
iii. Retrieve responses received to commands

The details of these functionalities have been described earlier.

4.5 Ground Interface

The ground interface module handles the exchange of data between the space segment and the ground station using CCSDS TC and TM packets.

4.5.1 Operation

On the uplink, data from the ground is received over the TT&C uplink interface in the form of CCSDS TC packets. The ground interface module processes the TC packets and extracts ground commands from them. Commands for the command execution module are passed into the command execution queues through the provided interfaces.

On the downlink, the ground interface module packages all the data that is to be sent to the ground station into CCSDS TM packets and sends it over the TT&C downlink interface. The telemetry data sent down are of two types. The first is generated by the various modules within the OBC and sent to the ground interface module for transmission. Besides this, during "Ground Station Pass' mode, the ground interface data periodically collects the latest real time telemetry buffer from the real time telemetry buffer maintained in the Telemetry collection module, and packages and sends this along with other data. The real time telemetry packaging and transmission is done once every second by the ground interface module.

The ground interface module maintains the following information that are updated during its operation.

i. Activities log. This log maintains information about the different mode changes, mission operations executed and faults during operation.
ii. Reports log. The reports log is a list of all reports generated in the OBC during operation. The report log also maintains the identifier of the report to be used for retrieving the report.
iii. Errors log. This is a list of all errors detected during operation.
iv. Offsets and sizes: This is the information regarding the sizes of the Telemetry buffers, Command queues and WOD file on the OBC and the offset which shows the last retrieved location by the ground.

These can be retrieved by the ground-station, or higher layers, for making decisions on operations.

4.5.2 Interface to autonomous operations modules

There is an exception in the operation of the ground interface module from the hierarchically layered architecture that is used in the flight software. All other layers in the system only communicate either with modules at their same level, or with lower layers. The ground interface module, however, initiates communication with the autonomy modules layer when commands to these modules are received.

The ground interface passes commands to the higher layer software directly to them, using the interfaces provided by them. This functionality can be implemented in a separate layer that is above the autonomy layers, however, for ease of development, this has been integrated into the common ground interface module.

PART-III (Autonomous Flight Software)

5. High Level Autonomy for Micro-satellite Operations

5.1 Paradigm for Operational Autonomy

The layers described in the previous chapters form the operational core of the flight computer. They provide the functionalities to command the satellite and the capability for the ground to use these functionalities. This framework, as mentioned earlier, supports the operation of the system in the traditional manner with minimal on board decision-making.

The layers for higher autonomy are built on top of this foundation. They use information and functionality provided by the lower layers the interfaces for the sensing of satellite state and execution of commands. This scheme enables overriding of operations planned on-board from ground station and also provides modularity by decoupling the autonomy modules from the command execution modules.

This chapter first establishes the paradigm that we will use for autonomous operations in the context of micro-satellites. It then gives an overview of knowledge representation and reasoning techniques that can be used for such missions. Following this, a description of two previous missions with a high level on board autonomy is presented.

The first step in planning for autonomy is to establish a framework and context for the system to work under and identify the characteristics of the operations and goals that the system would be carrying out. In the following sections, an attempt is made to define a generic Low Earth Orbit (LEO) micro-satellite mission characteristics and the system with respect to the flight computer.

5.1.1 The Environment

The operating environment of the fight computer consists of all entities around it that it interacts with it or affects its operation. These mostly consist of the other sub-systems on the satellite, the ground station, space and mission related areas of interest (earth / stars / earth or space based communication centres etc.)

The space environment is the operating environment affects the OBC operation in the sense that it has potential to damage or disrupt operations of the sub-systems (including the fight computer itself). The satellite environment consists of bus sub-systems, payloads and the various links on the satellite. The bus sub-systems would normally include an Attitude Determination and Control Service (ADCS) sub-system, communication link to the ground for Telemetry Tracking and Commanding (TT&C) and the Power Supply Unit (PSU) for power regulation and switching. Other sub-systems may be present for specific mission operations. The fight computer interacts with these using data messages to carry out the mission operations.

The ground station or the mission control station is the operating center for the mission. LEO orbits normally have a contact period of only three or four slots of about 10 minutes each every day. Commands for mission operation for a period of time are uploaded at one go during this period. Similarly, telemetry and mission data from previous orbits are downloaded during this slot. The speed of this link varies from mission to mission but they are the range is roughly between two to a few 10s of kbps for uplink and a few 10s to a few Mbps for downlink.

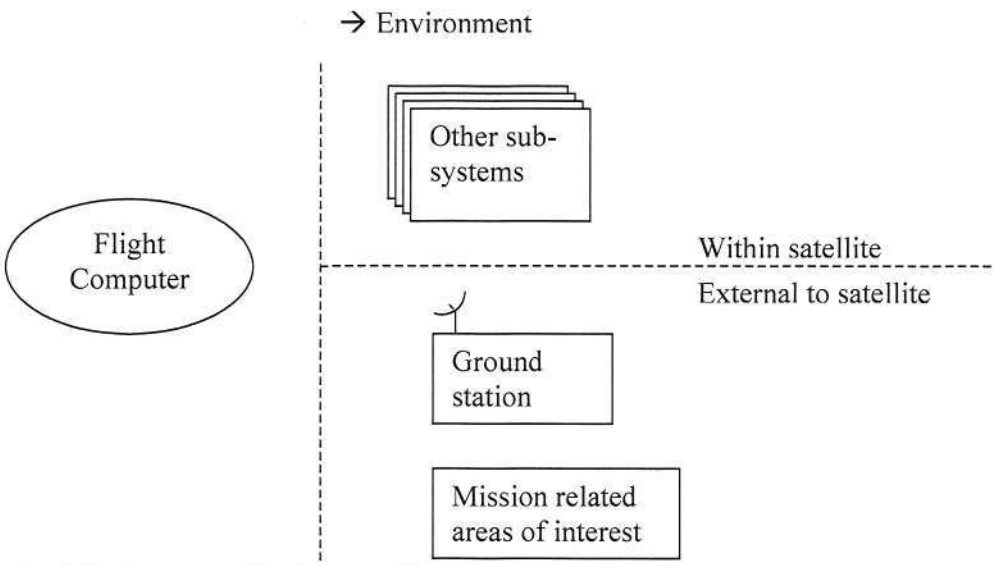

Figure 24 Flight Computer and its Operating Environment

The other objects of interest for the satellite are mission dependent and vary according to the characteristics of the mission.

The sub-systems on board the micro-satellite can be categorised into two types. Bus sub-systems and payloads. Bus sub-systems are non mission-specific sub-systems required for the satellite to survive and communicate with the ground station. This consists of the TT&C sub-system for ground communications, ADCS sub-system for attitude control and the PSU for power supply and switching. Besides this there would be one or more payloads that carry out mission specific operations. A typical operating environment for a micro-satellite with two payloads, PyL-1 and PyL-2 is shown in figure 25.

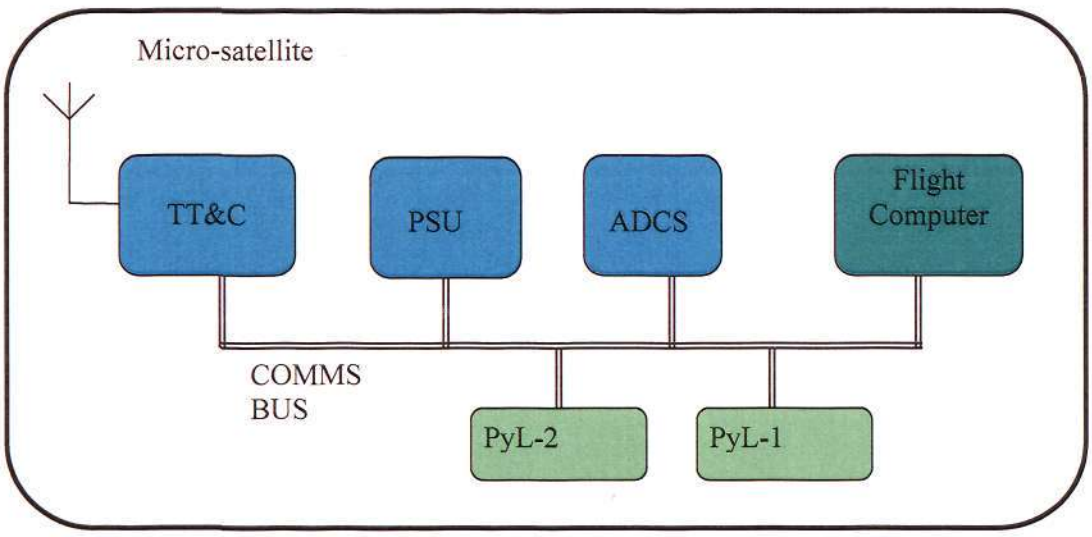

Figure 25 Typical Micro-satellite Environment

5.1.2 The Operations and Mission Goals

As described earlier, during the ground contact period, the ground station sends up a set of commands for the next window of operation. Typically they would consist of long sets of time or event triggered commands for individual operations, which would be executed at their respective times. The ground station would also request for telemetry of interest or mission data, if available, to be downloaded.

For a higher autonomy mission, the ground station only sends to the fight computer the mission goals for the next window of operation. The individual operations required for this operation are planned and scheduled on board. Similarly, once mission data or telemetry data that may be of interest are collected, the fight computer informs the ground of its availability and sends it down. This reduces the number of commands that

need to be sent to the fight computer as well as increases flexibility and robustness of the command execution. The fight computer can plan how to achieve the goal with the best resources available at the time of execution.

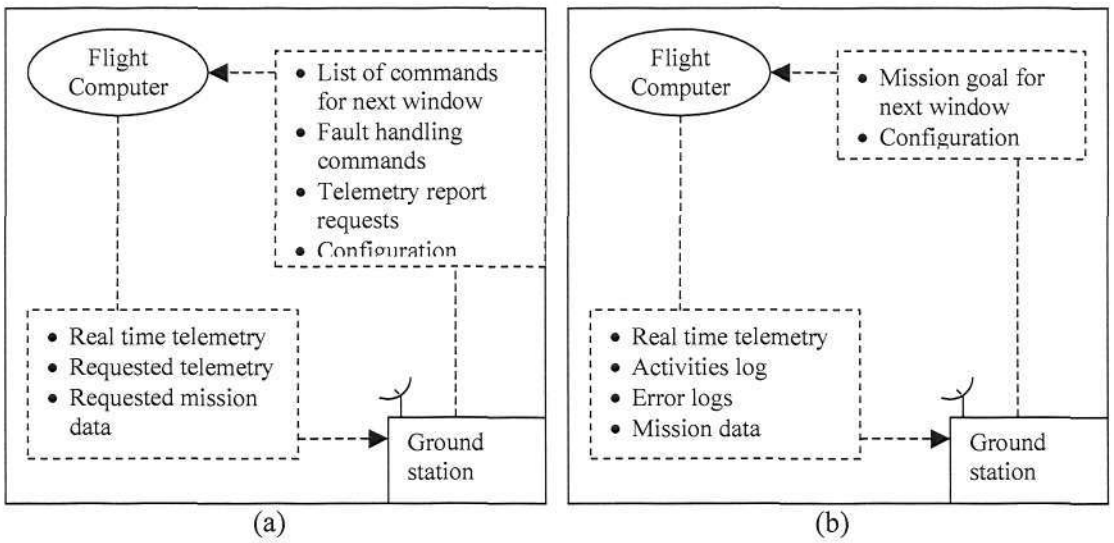

Figure 26 Ground-satellite interactions for normal vs autonomous missions

In usual satellite missions, fault handling involves only detection of the fault and going to 'Safe-hold' state. The ground station detects this, diagnoses possible faults and tries to restore the satellite to operational status. Besides the baseline operations, such as collection and monitoring of telemetry and execution of commands, normal operations carried out by an autonomous flight computer include inferring problems from detected anomalies in telemetry and taking appropriate fault-handling operations to return the satellite to operational mode. This decouples the ground from this operation and enables the satellite to get back to an operational state without having to wait for the next ground station contact. However, it is desirable for the ground station to be able to diagnose and attempt recovery in the traditional manner, overriding autonomous operations.

Exact mission operations and goals will depend on the mission. However, they generally involve operating the mission payload over specific areas of interest (or at times of interest) to collect mission data. The exact payload operation could be imaging, communications, science experiments or any other operation. However, to enable scalability and re-use of the design, the key operations involved for any mission goal is

the detection of area of interest and payload operation. The detection of area of interest requires a periodic sampling of the environment and monitoring for states matching the area of interest. The payload operation is then triggered. This would involve some checks to see if constraints and pre-condition for the operation are met. This would be followed by an initialisation of the payload and its operation, details of which would be mission specific. As in the case of autonomous fault handling, it is desirable for the ground to be able to monitor and override any operations planned on-board overriding the autonomous planning.

5.1.3 Information Integrity

In order to carry out its operations, the OBC would need to get information status information and sensor data from the different sub-systems on the satellite. Typically, this information is transferred to the fight computer through message interactions. Due to error-prone nature of the space environment, there is a high probability of error in the data received from the sub-system, in other words, corruption of the data in the message or failure in the link may result in incorrect information about the sub-system being inferred by the flight computer.

To minimise such situations, it is essential to do an integrity check on the information available to the flight computer to recognize, where possible, such errors and update the information accordingly. A scheme to do this has been developed for the OBC and will be discussed in the next chapter.

5.1.4 Some Considerations for Higher Levels of Autonomy in Operations

Conventionally, satellite missions are operated with a high level of operator interaction with the ground station. When moving from this approach to one where a higher level of autonomy and decision making power is placed on the satellite, several considerations and concerns have to be addressed. Prior to selection of the technologies for the high-level autonomy logic and design of the system, a study of the various considerations and technology options were done to help the trade-off between different schemes possible.

The key technical concerns and recommended methods to handle them are described here.

i. Increase in risk and drop in reliability of the system:
Increased software complexity generally results in a decrease in reliability. Therefore a high amount of complexity in the software algorithms, which are present to support autonomous operations, may potentially reduce the reliability of the system. This is especially true for highly critical operations such as fault detection and correction.

To address this, the overhead complexity introduced into the system to support autonomous operations must be limited. Higher performance does not necessarily mean higher complexity. The schemes used must be carefully studied to assess the implementation complexity and the traded off with the benefits introduced.

ii. Loss of mission or others due to inability to intervene:
If the mission is operated autonomous control by on-board logic, there is a possibility that there may be times where the software may make some mistakes in operation. This could lead to errors in functioning and even loss of mission because of erroneous behaviour of the software and the inability of the ground station to intervene in the operation.

The basic functionalities or lower level software should be kept possible to be commanded from both the ground and the higher-level software. To enable this, these should be de-coupled so that errors in the higher layers do not propagate through the whole system and corrupt it. Thus, both the ground and the higher layers control the satellite through a common lower level and ground inputs should always have highest priority. This ensures that the ground can (whenever feasible) always intervene and take over control.

iii. Cost and effort of development of system outweighing benefits:

This is more of a trade-off concern. If the cost and effort required to implement the system is higher than the net benefit derived from having on-board autonomy, this approach will not be sustaining one.

The key factors that will help reduce the cost of the system over in the long term are scalability and modularity. Scalability ensures that the system can be modified to suite different mission requirements as needed. Modularity enables re-use of previously developed modules and re-configuration of the system appropriately for the mission at hand. Even if initial costs are higher, this will help bring down costs over time.

iv. Difficulty in validation, simulation and diagnoses of system on the ground:
Autonomous operation may use Artificial Intelligence (AI) technologies for their software. One of the intrinsic properties of several of these approaches is their unpredictability. This is true of learning algorithms are used on board for any processes. This would mean that validation of the software before the mission would be more difficult than for traditional missions. Moreover this also means that at a later stage during the mission, the internal state of the software may not be known to the ground. Thus, simulation of the system on the ground, to study and diagnose errors may not be possible.

The use of learning algorithms and techniques is the main factor that contributes to this, so limiting the use of such techniques will help reduce this problem. The schemes used for designing the internal logics of each of the modules are the issue here and they should be evaluated carefully to minimise risks of this kind.

The architectural features address issues (ii) and (iii). Proper selection of methods to implement the software autonomy help to remove concerns (i) and (iv).

Based on the above discussion, the following key characteristics and desirable features can be identified in the system.

i. Limited action space.

The set of operations available for the flight computer are limited to the number of different commands available to operate the various sub-systems.

ii. Need for deterministic operations

In order for the ground station to be able to take over control/diagnose problems of the satellite if and when required, it is desirable for the satellite operations to be deterministic and predictable. This would limit the capability to of the system to learn and evolve in a major way over time.

iii. Need to verify information integrity

As discussed earlier, the error prone nature of the environment makes it necessary, wherever possible, for the flight computer to check for and verify the integrity of the information it has about the environment.

iv. Support to override autonomous operations from ground station

For reliability reasons, it is desirable for the ground to be able to override autonomous operations and operate the satellite only under ground commands, whenever it wants. The ground should thus have highest priority at all times.

v. Scalability and re-usability

In order for development to be cost efficient in the long term, it is necessary for the design to be abstracted from mission specific details and scalable to different missions.

vi. De-coupling of autonomy modules from baseline operations

The autonomy software would naturally be more complex than the baseline operations software; since software complexity and reliability are inversely proportional, it would therefore be more likely to develop errors. In order to prevent propagation of faults through the whole system and protect the core software, it is necessary to de-couple and abstract the autonomy modules from the core software.

5.2 Approach for Autonomy

The flight computer is required to operate the satellite to achieve certain specific goals. This involves generating the individual commands for the baseline command execution layer to achieve the mission goal. The task of generating a sequence of actions that will achieve a certain goal is known as planning. For an entity to carry out this kind of

activity, it is necessary for it to have an internal representation or knowledge about its environment and actions that it can take. Based on this, there must be an algorithm to reason as to what actions can produce the required goal.

The first aspect of this is a Knowledge Representation problem. The 'capability to reason with the representation' is the algorithm and methods used by the agent to search through possible actions and identify the most optimum sequence that achieves the goal.

5.2.1 Schemes for autonomous mission planning

One approach to represent the world for planning is the simple state space method. This involves representation of the various possible states of the system as a state space, with the connectivity between states representing the action required to take the system from one state to the next. The planning task is thus a search operation within this state space connecting the initial state to the required state. This scheme, however, does not scale well as the state space increases in size or changes. Also it is difficult to make a complete state space capturing all possible transitions of the system.

An alternative planning method is to represent the system as a logical model and use logical reasoning with this model for planning.

5.2.2 Knowledge Representation and Logic based Planning

Knowledge representation involves creation of a *knowledge base* of the environment to reason about actions and expected changes to the environment. This can be done in any way that captures the required information. A common method used to capture the information is using a set of logical sentences expressed in some logic system. The sentences form the information in the knowledge base. There also needs to be in place, a system of reasoning, based on the logic system used, to solve problems. The logical language provides the syntax and semantics of the sentences in the knowledge base. *Syntax* refers to the legal form of the sentence and *semantics* defines the truth or false of the sentence. A world thus created consisting of all the sentences in a knowledge base is called a *model*.

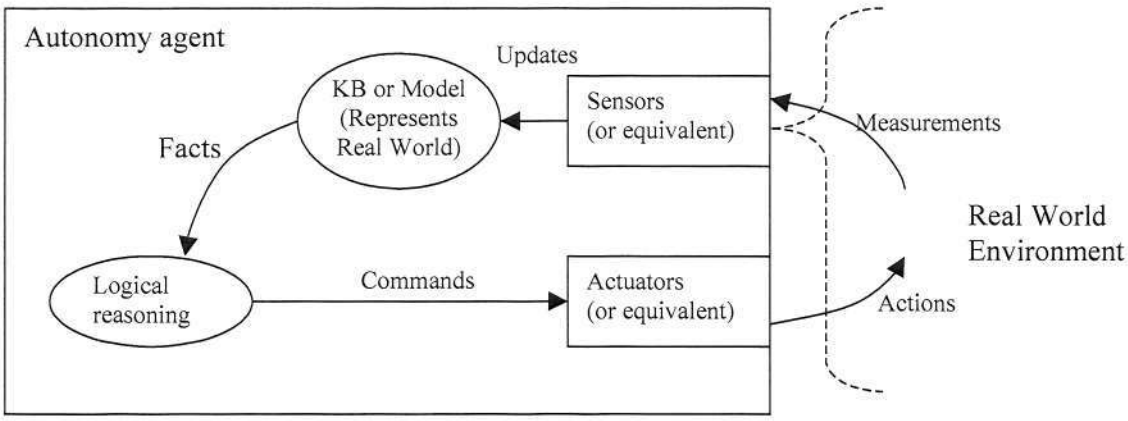

Figure 27 Agent uses KB or Model to reason about environment

Axioms refer to information that is given in the knowledge base. Theorems are information that can be inferred from the axioms in a knowledge base. Establishing the truth of given information with respect to a model is called Theorem Proving. Any information, represented by a sentence, is valid in a model if it can be proved to be true in a model, i.e., the sentence logically follows from the information in the knowledge base. Proof of logical sentences can be carried out (inferred) using valid inferential rules in the logic system being used; or using other techniques such as model checking.

5.2.3 Propositional Logic

Propositional logic is a simple logic system that deals only with facts. Facts in propositional logic are represented by propositional symbols that are either true or false. The symbols True and False themselves have special meanings, which are true and false respectively. Tables 2 and 3 below give an overview of the syntax and semantics of propositional logic.

Symbols	P | Q | True | False …
Atomic Sentence	True | False | Symbol
Connectivity	^ and/conjunction v or/disjunction => implies/implication. <=> equivalence/biconditional ~ not/negation.

Complex Sentence	Sentence v Sentence | Sentence ^ Sentence | Sentence => Sentence | Sentence <=> Sentence | ~ Sentence

Table 2 Propositional logic syntax

P	Q	~P	P^Q	PvQ	P=>Q	P<=>Q
False	False	True	False	False	True	True
False	True	True	False	True	True	False
True	False	False	False	True	False	False
True	True	False	True	True	True	True

Table 3 Propositional logic semantics

Propositional logic is thus a very simple logic that can be used to represent the world, however, it lacks expressive power required to represent relations and functions in real world situations.

An example representation of the satellite system using propositional logic is given below.

Let the following symbols mean as follows

PWR_AVAIL_TX: There is sufficient power available for data transmission.

GSTATION_ INRANGE: The satellite is above the ground station.

DOWNLINK_ACTIVE: The downlink is active.

DOWNLINK_ACTIVE <=> PWR_AVAIL_TX ^ GSTATION_INRANGE represents the fact that the downlink is active if and only if there is sufficient power for transmission and the satellite is above the ground station. Similarly all facts and relationships can be expressed as symbols and entities.

5.2.4 First Order Predicate Logic

First Order Predicate Logic(FOPL) or just First Order Logic (FOL) extends the expressive power of propositional logic with functions and relations. FOL consists of objects, relations and functions. Objects represent entities in the world. Eg. Boy, Girl, Apple. Relations can be unary functions or properties or n-ary relations or properties. Eg. Bigger than, smaller than, etc. Functions are relations with only one value for one input. Eg. Father of, next in line etc.

The FOL model consists of symbols, which can be constant symbols, predicate symbols and function symbols. *Constant symbols* represent objects. *Predicate symbols* represent relations and *function symbols* represent functions. A *term* in FOL is a logical expression referring to an object. The table 4 below gives an overview of the syntax for FOL.

Constants	John | Apple | Boy | …
Variables	x | y | …
Predicate	Before | After | Bigger than | ….
Function	Father of | LefLeg | ..
Term	Function(Term…) | Constant | Variable
Atomic Sentence	Predicate(Term,…)
Connectivity	^ and/conjunction v or/disjunction => implies/implication. <=> equivalence/biconditional ~ not/negation.
Quantifiers	∃ existential ∀ universal
Sentence	Atomic Sentence | ∃ variable,… Sentence | ∀ variable,….Sentence | Sentence v Sentence |

	Sentence ^ Sentence |
	Sentence => Sentence |
	Sentence <=> Sentence |
	~ Sentence

Table 4 FOL syntax

In FOL, variables and quantifiers can be used only for objects, not relations or functions. A *Well Formed Formula (WFF)* is a legally formed sentence. Facts or information in FOL are represented by WFFs. Establishing the validity of a WFF in a model can be done using the inference rules provided by FOL. A sentence is said to be *grounded* if all its variables are replaced by objects in the real world or constants, ie, there are no variables. An *interpretation* of a model is a mapping of constant symbols, predicate symbols and function symbols to objects, relations and functions in the problem domain.

An example representation of the satellite system using first order predicate logic is given below.

Let the following predicates represent the follows

Pwr_Level_Suff(x): There is sufficient power available for operating 'x'.

In_Range(x): Satellite is in communications range of 'x'.

Is_Active(x): The sub-system 'x' is active.

And the following objects represent the follows

Transmitter: The TT&C transmitter

Ground_Station: The ground station

Is_Active(Transmitter) <=> Pwr_Level_Suff(Transmitter)^ In_Range(Ground_Station)

As in the case of propositional logic, this represents the fact that the downlink is active if and only if there is sufficient power for transmission and the satellite is above the ground station. However, in the case of predicate logic, adding the necessary objects makes it possible for the same predicates to be extended to cover information regarding them as well. If Sub_Systems I a set of objects consisting of all sub-systems on the satellite, then

$\forall$ x ϵ Sub_Systems Is_Active(x) => Power_Level_Suff(x)

represents the fact that for all sub-systems, the sub-system is active only when the power level is sufficient for its operation.

FOL thus has a much higher expressive power than propositional logic and can model real world information more realistically. The OBC knowledge base is based on irst Order Logic. Other logics that have even higher expressive power include Higher Order Logic and Temporal Logic.

5.2.5 Planning using Situational Calculus

Actions are logical terms that, if executed, can change the information in the knowledge base. The planning task involves using an inference algorithm that finds a set of actions that will change the situation from the initial situation to the required situation. In *Situation Calculus*, situations are logical terms representing the initial situation (normally S_0) and all situations that are generated by applying an action to the situation. The function Result(a,s) represents the new state when action a is executed in situation s. *Fluents* are functions and predicates in the model that vary from one situation to another.

An action is described in the following manner:

Preconditions => Poss(a,s)

Poss(a,s) => Changes to state when action a is exectuted.

Thus, the preconditions and effects of all actions available are described.

However, this scheme leads to something called the *Frame Problem*. The action only describes the changes to the model, what remains the same is not described. This is a difficulty for the formal proof process that goes on to establish the validity of the required situation. There are schemes developed to overcome this problem (successor state axiom), however, they involve a substantial increase in the knowledge base size and reduce the efficiency of the system.

5.2.6 The STRIPS Planning Language

One of the classic planners is the STRIPS planning scheme. STRIPS stands for Stanford Research Institute Problem Solver and was developed in the 1970s.

STRIPS uses a combination of Means End Analysis and Theorem Proving for generating a plan. A state specification, consisting of a set of logical expressions signifying the current state, represents the world. The basic idea that underlies Means-Ends Analysis is that the planner has the ability to compare two problem states and determine one or more ways in which these problem states differ from each other. Based on the differences, it recursively plans the actions that it must take to move from the initial state to the goal state. Theorem Proving, as described earlier, is used to establish the truth of a given information with respect to the state specification.

The STRIPS planner works with the *closed world assumption*, any information not given in the knowledge base is assumed to be false. The knowledge base is assumed to be complete. Actions that the planner can take are defined in terms of preconditions and effects. Preconditions in the current state are checked using theorem proving. The effects are described in terms of '*add*' and '*delete*' lists. The add list is a set of information that must be added to previous state, the delete list is the set of information to be deleted from the previous state.

The effects of an action is thus represented as a difference from previous state rather than an effect statement. This avoids the frame problem faced in situational calculus. STRIPS has the versatility and scalability required to make it reusable for multiple missions, and at the same time is simple enough to be used to represent the problem and capture the planning efficiently. The mission planer for the OBC will be developed using STRIPS. A detailed description of STRIPS and the planning algorithm for the OBC is given in the next chapter.

5.2.7 State Update & Data Integrity Verification

One of the assumptions in the classical AI planning framework is that there is only one causal agent in the system, and it is the planner. Thus, the planner has complete control over the system and changes happen only when the planner takes an action. Unfortunately, in the case of most real world systems, this assumption is not true and especially so in the case of an error-prone environment of a satellite mission.

In order for the planner to make optimum plans, it must have access to the as accurate a representation of the system as possible. The model of the world used for the reasoning and planning thus has to be periodically updated based on telemetry and status information available about the system. The OBC gets information regarding the environment through data messages from the other sub-systems. Based on the data received from the different sub-systems, the knowledge base is updated. This updated knowledge base is then used for fault handling and mission planning operations.

5.2.8 Scheduling

Real world planning problems include planning and scheduling of planned actions. Scheduling is the allocation of available resources and ensuring that time and state constraints are met by the plan. The approach chosen to tackle planning and scheduling depends on the complexity of the planning task and the world. Typically, these are done using dedicated for one or the other for highly complex tasks and worlds.

An alternative approach is to represent the scheduling information along with the state information as literals. This includes representational and reasoning capability for scheduling as well into the planner. This is the approach that was taken for the OBC mission planner. Details regarding its design are described in Chapter 7.

5.3 Fault Diagnosis

Due to errors in the sub-systems or data corruption in the links, sub-systems may become unavailable or develop errors in their operation. Sometimes, taking certain steps can help correct the error, other times, the sub-system may become lost permanently and

appropriate steps have to be taken to plan/execute the rest of the mission without the damaged sub-system. In order to take the correct recovery action, it is required to detect and diagnose the fault and isolate the potential cause. Together this constitutes the fault handling operation on the OBC.

5.3.1 Possible Schemes

Research in the field of AI has resulted in the development of several methods for achieving higher amount of intelligence and diagnostic capability in software systems. This section gives an overview of some of the possible methods for the design of a diagnostics module.

i. Rule based expert systems

'Expert system' is the term used to refer to knowledge-based programs that provide "expert quality" solutions to problems in a specific domain. The heart of an expert system is the knowledge base used to generate the solution. In a Rule base expert system, the knowledge representation is in the form of a set of 'IF...THEN...' rules. An inference engine applies these rules to the data from the problem domain and attempts to generate the solution.

The rules in the knowledge base of the system are of the generic form 'IF <antecedent> THEN <consequent>'. When the condition specified in the antecedent is true, i.e. if there is a match between the antecedent and some problem domain data, the consequent is executed and the rule is said to be 'fired'. The inference engine thus cycles through MATCH-FIRE cycles based on the data and the knowledge base rules. Besides the inference engine and the knowledge base, a Rule base expert system typically also has an explanation sub-system. The explanation sub-system provides the user information about why a certain solution was generated. This is done by keeping track of the inference steps that resulted in the solution.

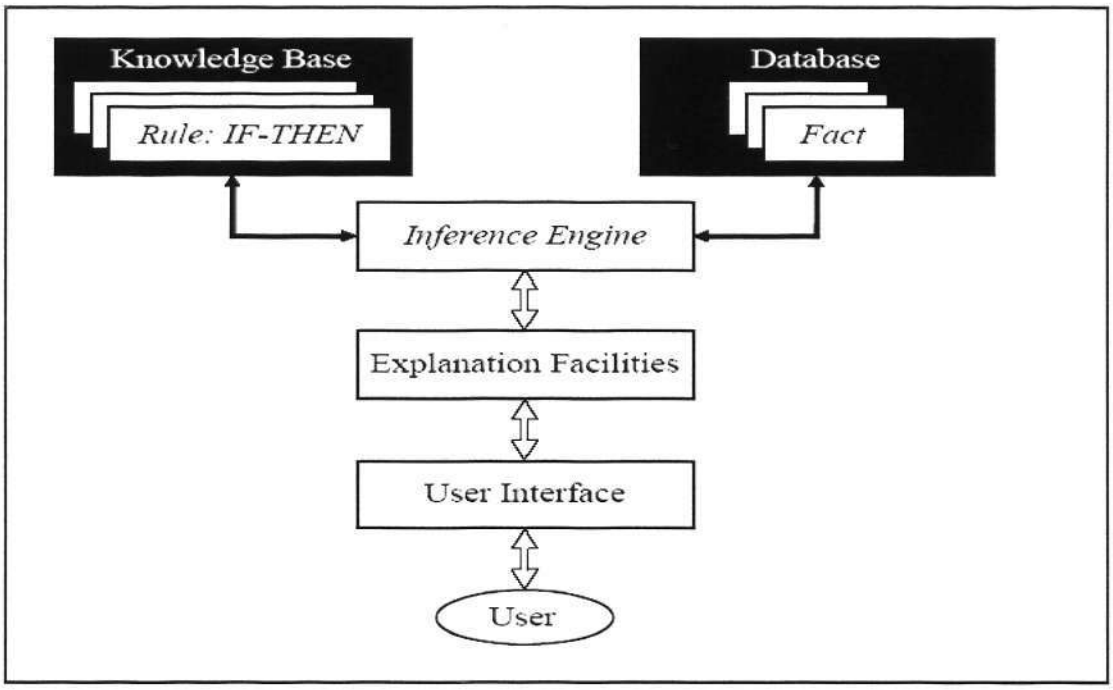

Figure 28 Structure of a Rule based expert system

Some of the key features of such a system are

- Separation of knowledge base from inference engine and user interface. This is a very useful property as this enables re-use of the inference engine for different problem domains
- Narrow area of focus
- Employ symbolic reasoning for solving a problem
- Do not have the capability to learn
- Easy to modify by adding rules into the knowledge base
- Opaque relations between rules. Although the individual rules are simple and easy to evaluate, the relationships between the rules in the context of the system as a whole is not easy to observe
- For systems with a large number of rules, the time taken for processing the rules before a solution can be available may be too long for real time systems

ii. Case Based Reasoning

Case-based reasoning (CBR) is a reasoning scheme that uses specific knowledge of previously experienced problem situations or cases to solve each problem. This is fundamentally different from other major AI approaches because instead of relying solely on general knowledge of a problem domain, or making associations along generalized relationships between problem descriptors and conclusions, CBR is able to utilize the specific knowledge from previous experience. A new problem is solved by finding a similar past case, and reusing it in the new problem situation. Each new experience is retained each time a problem has been solved, making it immediately available for future problems. If an exact match is not found in the past cases set, adaptation techniques may be used to find a near match or the best-fit solution.

The decision on how to represent and store each case data should be addressed specifically targeting the problem domain and selecting relevant data. In general, the content of a case is made up, the problem/situation description, the solution, and the outcome. The outcome is not needed but could be added to suggest solutions that work and use cases with failed solutions to warn of potential failures. Indexes predict a case's usefulness. Indexes should be made that describe the tasks a case can be useful for. Indexes should be abstract enough to retrieve a relevant case in a variety of future situations, and indexes should be concrete enough to be easily recognizable in future situations.

Case-based reasoning is - in effect - a cyclic and integrated process of solving a problem, learning from this experience, solving a new problem and so on. CBR is an approach to incremental, sustained learning, since a new experience is retained each time a problem has been solved, making it immediately available for future problems.

Key features of CBR:

- A very important feature of case-based reasoning is its coupling to learning. Learning in CBR occurs as a natural by-product of problem solving. When a problem is successfully solved, the experience is retained in order to solve similar problems in the future.

- CBR allows the system to propose solutions to problems quickly, avoiding the time necessary to derive those answers from scratch
- Cases can warn of the potential for problems that have occurred in the past, alerting the system to take actions to avoid repeating past mistakes
- Cases may not be representative of the domain and may not cover the domain well. A few exceptional cases can affect the performance of a CBR based system.

iii. Model Based Reasoning

Model Based Reasoning (MBR) is an inferring process using models abstracted from the reality of a physical system for reasoning. MBR is the symbolic processing of the representation of the working of a system in order to predict, simulate and explain the resultant behaviour of the system. Any system developed by using Model Based Reasoning techniques is broadly called a Model Based System.

The fault-handling scheme on the OBC is based on a case based reasoning approach. This enables a database of fault scenarios that can be built up over time thus enhancing the autonomous fault handling capability over time.

PART-III (Autonomous Flight Software)

6. Autonomous Operations Software

6.1 Overview

This chapter describes the design of the higher-level autonomy software layer of the flight computer. First, an overview of the autonomous operations targeted and the modules in this layer is presented. The Mission Planner agent, which is based on the STRIPS planning language, is then described. Following this, the model used to represent the flight computer environment is discussed. A dedicated agent is responsible for updating and maintains the X-Sat internal model from telemetry and status. Lastly, the Fault Handler agent on the flight computer is presented.

The autonomous activities to be carried out by the flight computer during autonomous operation mode of the mission are as follow.

i. When powered on, the flight software initialises the satellite and verifies sub-systems.
ii. Ground station sends or has preset high-level mission goals for the system, eg. Image specific areas whenever possible. The flight software generates and executes appropriate plan(s) to achieve this goal, based on the constraints and resource availability.
iii. When over the ground station, the flight software downloads all mission related data, real-time telemetry and items of interest such as error reports, WOD and activities log.
iv. The flight software detects faults in operation and takes recovery actions.

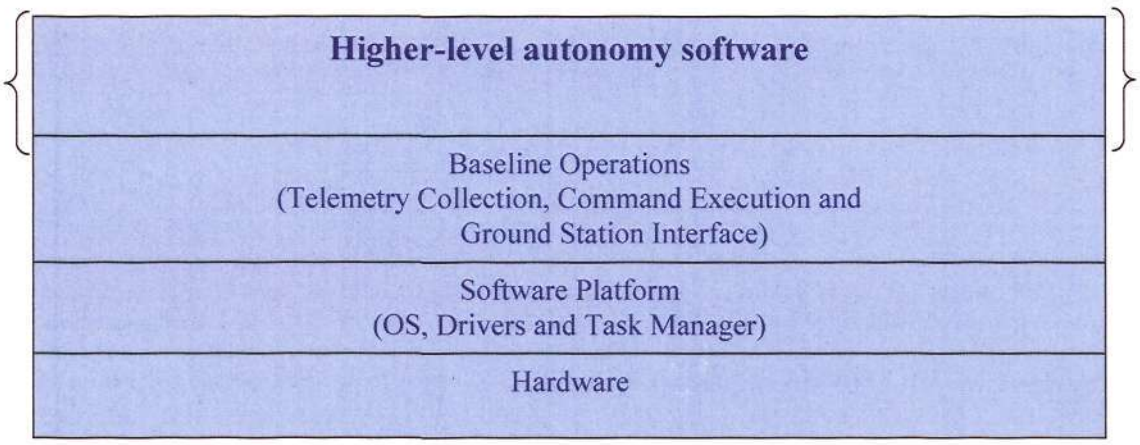

Figure 29 Autonomous Operation Layer in the Context of the Flight Computer Layers

The autonomous activities that need to be carried out on the flight computer to enable operation have been partitioned into three categories. These are

i. Mission planning,
ii. Telemetry monitoring & state update, and
iii. Fault handling.

Each of these activities is implemented as autonomous software agents in the flight computer. This layer thus consists of three distinct software agents that can be enabled or disabled through ground command.

The mission planner receives high-level goals from the ground station and breaks them into individual commands required for the operation. Once the plan is generated the plan generated, the planner monitors the state and releases each action (command) in the plan as they become ready for execution.

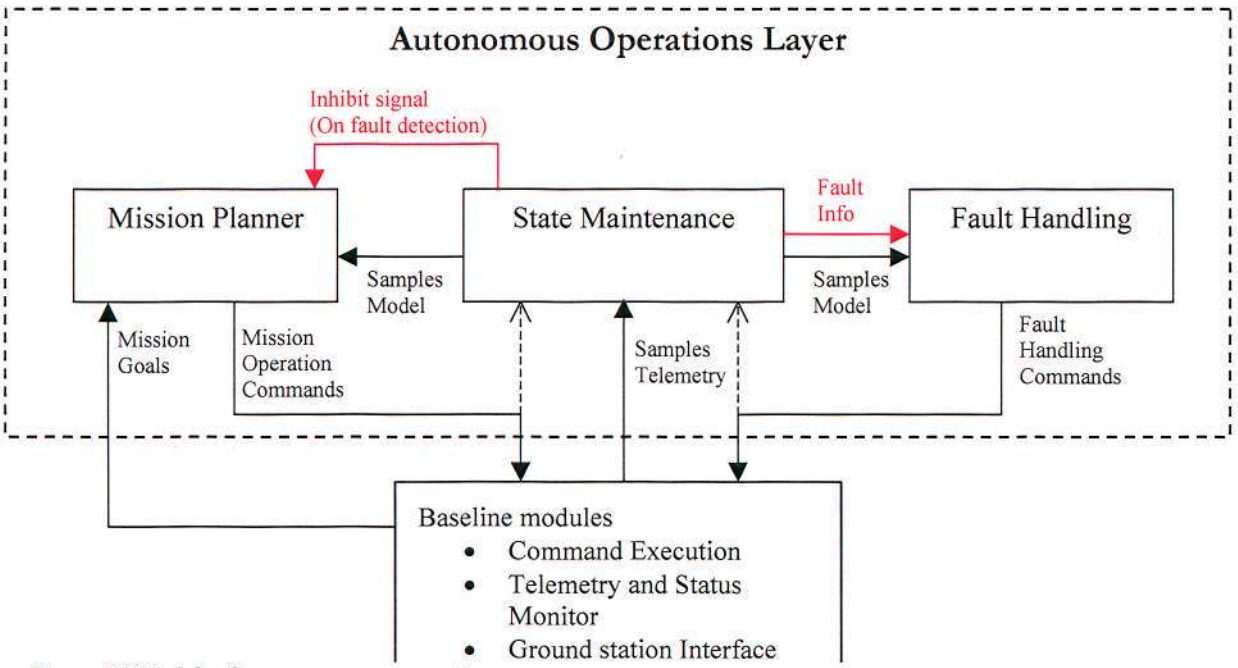

Figure 30 Modules for autonomous operation

The figure above is a block diagram that gives an overview of the different modules and their interactions.

The state maintenance module continuously collects the received telemetry and status information and update the state specification of the system that is maintained within the flight software. This module is also responsible to detect any anomalies in the received data and trigger fault handling operation, on detection of any faults.

The fault handling operation, once triggered, attempts to diagnose the fault, take appropriate recovery operations to prevent any further faults and try to return the system to an operational state. On detection of faults, the mission planner is inhibited by the model maintenance agent.

6.2 Mission Planner Agent

This section first goes over the STRIPS planning language in more detail. It then describes the design of the autonomous mission planner.

6.2.1 The STRIPS Scheme

Model: The STRIPS language uses a model of the world to represent world states and for planning. The model is based on First Order Logic and consists of a conjunction of positive literals to represent the state. Literals in the state description must be ground and function free. Eg. IsOn(Antenna_1) ^ IsTransmitting(TTC). The closed-world assumption is used with the model, so the description should capture ALL relevant information in the world.

Goal: The goal in STRIPS is a partial state represented by a Well Formed Formula (WFF) that is a conjunction of positive, grounded and function free literals. The model in a particular state satisfies the goal if the state contains all the atoms in the goal. The literals in the goal state are assumed to be independent of each other.

Actions: The STRIPS knowledge base also includes a list of actions that can be executed in the world. The action is described using variables that can be instantiated using appropriate constants. This way of describing the action is called an action schema.

Each action is specified in terms of preconditions that must hold for it to be applicable and the effects that ensue when it is executed. The precondition is a function-free WFF of ground literals that represent what must be true before the action is applicable and can be executed. As in the case of the goal, the precondition is satisfied if the state contains all its atoms.

The effect is a conjunction of function-free ground literals that describe how the state changes. Positive literals in the effect are added to the current state and negative literals are removed from the current state. If a literal in the effect is already true in the current state, it does not have to be added or deleted. They can also be specified separately using only positive literals in separate add and delete lists. The state resulting after adding the add list and removing the delete list is the result of the action. Any literal not mentioned in the add or delete list stays the same.

Example action schema

Action(Operate (x),

PRECOND:	IsOn(x)^IsOk(x)^HasPower(x)
ADD EFFECT:	IsOperating(x)
DELETE EFFECT:	IsIdle(x))

Proof: The process of establishing the truth of a sentence from a given state representation is the proof of the sentence. Proof in STRIPS is carried out using a resolution theorem prover.

The resolution theorem prover first negates the theorem or sentence to be proved and adds it to the state specification, which is known to be true. It then uses the Resolution Rule of inference to show that this leads to a contradiction. A detailed description of the resolution rule is not covered here.

The STRIPS resolution theorem prover operation uses a search through the specification model and for the sentence that it is trying to prove. If the literals in the sentence exists in the specification, it is true, if not, it is false. Thus, the closed world assumption is utilised.

Planning: Once the knowledge base is established, the planning task is a search problem within the knowledge base to identify the action sequence required for the goal. The search operations searches through the representation of the system until it finds a path between its initial point and final point. There are various approaches to search problems. During the search, each intermediary state may have several other states it connects to. A search routine that checks each of these fort the goal before going to the next state from any of them is called a Breadth First search. The alternative to this is Depth First Search where the routine chooses one of these states, checks it for the goal, chooses one of the states connected from this state, checks it, and so on until it finds a state with no further states to go to, and repeats this process for each of the possible paths until it finds the goal or covers all states.

Another way of classifying the search is based on direction of search. A search that starts at the initial state and searches for the goal is called a progressive search. One that starts at the goal and searches backward to reach the initial state is called a regressive search. Pure depth or breadth first searches with no other information are brute force techniques. These become very inefficient as the problem domain gets larger. An alternative to brute-force search is to use a heuristics for the search. In this case, the search routine is provided with certain other parameters or information to help select the order in which to expand the paths. Good heuristics greatly help reduce the search operation time.

In the case of STRIPS, the planner tries to prove the goal with the current state. If the goal is not provable, the planner looks for actions that have effects that make the goal true, in their respective add lists. The preconditions for this action becomes the next sub-goal and the operation is repeated. The STRIPS planning operation recursively uses the model and the goal to identify a set of actions that will result in the goal being satisfied.

This process starts with the main goal and carries on recursively until the goal is satisfied. The output of this process is a list of instantiated actions that will achieve the goal.

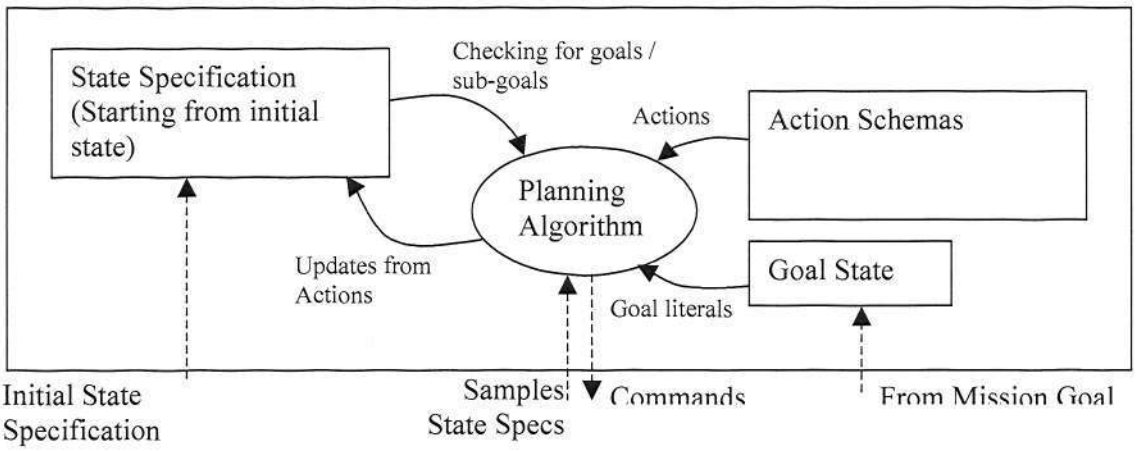

Figure 31 STRIPS Based Planner

Besides the current state representation, the final goal and the action schemas list, the planner uses some other data structures as well for its operation. These are the goal stack, and the plan action stack. The goal stack is a stack into which each consecutive subgoal WFF identified is pushed during the operation. The topmost goal in this stack is the current goal. The plan action stack consists of the set of ordered actions selected by the routine as it identifies actions with the effects required to reach the goal state.

6.2.2 The Flight Software Internal Knowledge Representation

The various functions, predicates and objects that form the flight software knowledge base, and what they represent in the real world, are described in Tables 5 and 6.

Predicates

Predicate	Real World Information
IsOn(x)	Sub System 'x' is switched on
IsBus(x)	Sub System 'x' is a bus sub system and not a payload
IsPayload(x)	Sub System 'x' is a payload
IsAvailable(x)	Sub System 'x' is available for use and not damaged
NotAvailable(x)	Sub System 'x' is not available for use
IsReady(x)	Sub System 'x' is ready for operation

PowerSuff(x)	Power level is sufficient to operate Sub System 'x'
PowerInsuff(x)	Power level is in sufficient to operate Sub System 'x'
RTTMEnabled	Real time telemetry downlink is enabled
ImageDataAq FALSE	Image Data Not Acquired
ActivityLogDnlded	Activity Log downloaded to ground
FaultReportsDnlded	Fault Reports downloaded to ground
ImageDataAq TRUE	Image Data Acquired
ImageDataDnlded	Image Data Downloaded to Ground
MTDataAq FALSE	ADAM Mobile Terminal Data Not Acquired
MTDataAq TRUE	ADAM Mobile Terminal Data Acquired
MTDataDnlded	ADAM Mobile Terminal Data Downloaded to Ground
FaultReports TRUE	Fault Reports present on board
FaultReports FALSE	No Fault Reports present on board
GSInRange TRUE	Ground station is in communications range
GSInRange FALSE	Ground station not in communications in range
DiskNotFull	RamDisk is not full
IsTime(t)	Time value is 't'
Downlink On	Downlink active
IsAcrive XBT	X-Bank transmission link active
IsLocation(l)	Location is 'l'
IsMode(m)	Current FLIGHT SOFTWARE mode is 'm'
IsADCSMode(m)	Current ADCS mode is 'm'
InitialisationDone	System initialisation has been done

Table 5 Predicates used in the model representing the satellite

Objects

Object	Real World Information
OBC	Current OBC (Self)
TTC_TX1	TT&C transmitter 1
TTC_TX2	TT&C transmitter 2

TTC_RX1	TT&C receiver 1
TTC_RX2	TT&C receiver 2
PSU	Power Supply Unit
ADCS	ADCS sub-system
XBT1	X-Band sub-system 1
XBT2	X-Band sub-system 2
Iris	IRIS imaging payload
ADAM	ADAM communications payload
PPU	PPU data processing payload
GPS_A	Accord GPS Receiver
GPS_P	Phoenix GPS Receiver

Table 6 Objects in the model representing the satellite

The symbols defined here provided the literals to model the various states of the system.

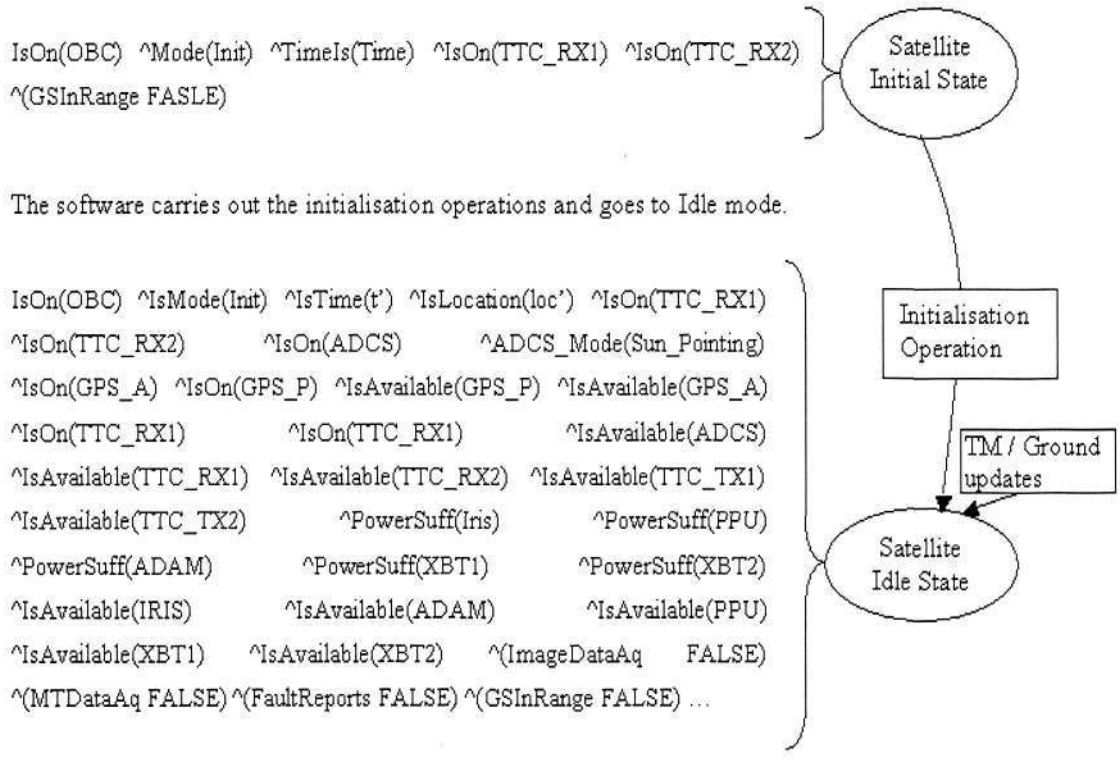

Figure 32 OBC Initialisation

Here, the entries t', loc' and att' are the current time and location information. These are updated from ground or through telemetry collected. This represents the satellite in Idle state, with no faults.

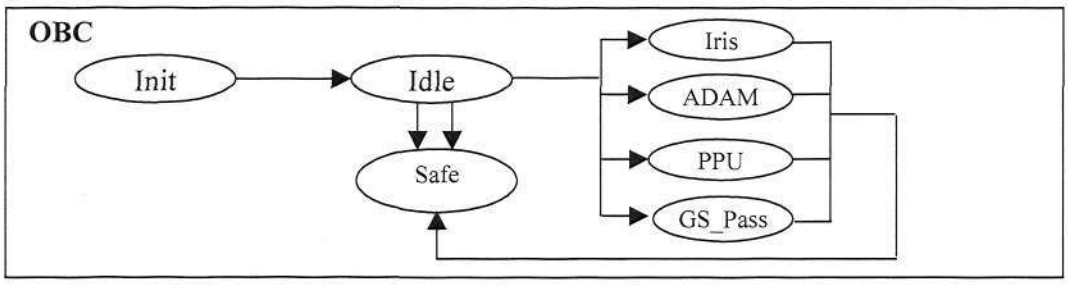

Figure 33 Different Modes of Operation (States) of the OBC

6.2.3 Mission Planning

The inputs to the mission planner operation are goals from a goal list, which is checked periodically by the planning algorithm. Each entry in the goal list represents a mission goal, its trigger and priority. The goal controls the activities that the planner plans. The trigger schedules the goal, i.e., the planner generates a plan for the goal only when the trigger condition is reached. Besides this information, each entry also has a priority number. The planner uses the priority number to choose the goal to plan for, when multiple goals are triggered at the same time. The enable status information signifies if the goal is valid or not.

In the present version, the trigger is done the instant the operator would want the actual operation to start. This however may not be feasible in a realistic scenario and has to be modified later.

Scheduling and Area of Interest Database: The trigger information for scheduling of the goal can be based on either time, or any event or state that can be represented from predicates allowed for system representation. The planner checks the goals list once every second, to provide a time resolution of 1 second.

An alternative method of scheduling mission goals is through an area of interest database. The area of interest database is a mapping from geographical location to a binary '1' or '0' representing if it is an area of interest or not. This database would be implemented as part of the model in the state specification. The LISP implementation of the model used for evaluating the planner uses such a scheme.

Table_ID: ADAM_AOI

Location (+/- 2)	AOI
(1, 103, X)	1
(2, 103, X)	0
(3, 103, X)	0
(4, 103, X)	0
(5, 103, X)	0
..	..

Figure 34 Sample Area of Interest Database

The resolution (1 here) and margin (+/- 2 here) can be specified according to the specific operation and requirements. The trigger would thus be [ADAM_AOI]. The planner interprets this as 'if the Location information has a 1 for AOI entry, trigger the operation'. This scheme also enables scheduling of multiple operations over areas of interest, if constraints are satisfied, thus maximising the payload operation and mission data.

Mission Goals: Each mission goal identification consists of a goal, a priority number between 0 – 255 and a trigger criteria. Mission goals can be classified into two kinds of goals. Pre-programmed goals and ground commanded goals. The lower the priority numbers the higher the priority of the goal. Some of the possible goals that can be used are listed below.

Certain pre-programmed goals can be built into the design. These are

i. [Initialise] [0] [IsOn(OBC) ^ Mode(Init)]

The first goal is to check all the bus SubSystems telemetry points verify them.

ii. [Initialise_ADCS] [2] [IsOn(OBC) ^ Mode(Init)]

The next goal on power up is to detumble the satellite and bring it to a sun pointing mode so that the battery can be charged optimally.

iii. [GS_Downlink] [50] [GSInRange]

This goal is the ground station pass operation and is triggered when the ground station is in range

The ground commanded goals identified for the OBC design are

i. [Iris_Op] [priority] [trigger]

This goal schedules IRIS payload operation.

ii. [ADAM_Op] [priority] [trigger]

This goal schedules ADAM payload operation.

iii. [PPU_Op] [priority] [trigger]

This goal schedules PPU payload operation.

[Goal]	**[Priority]**	**[Trigger]**
[Initialise]	[0]	[IsOn(OBC) ^ Mode(Init)]
[Initialise_ADCS]	[2]	[IsOn(OBC) ^ Mode(Init)]
[GS_Downlink]	[20]	[InRange[Ground_Station] ^DataAcquired(Iris_Image)]
[IRIS_Op] [Pos, Att]	[2]	[TimeIs(21JAN05 125900) ^Mode(Idle)]
[Adam_Op]	[5]	[ADAM_AOI]

Table 7 Sample Goal List

Table 7 shows a sample goal list. The goals in the goal list entry must not be confused with the STRIPS goal state. The goal here is a mission goal, which tells the planner what activity to plan for. The goal is a macro representation for a set of sequential STRIPS goals and individual triggers, which the planner expands it out into. The expansion of goal to a macro is done on board with the help of a look up table, with entries for each of the possible goal macros that the planner can handle. Each goal macro expansion consists

of a sequential list of individual STRIPS goals, with their respective triggers. The expansion of the goal macros listed above, in terms of STRIPS goals, which the STRIPS planner tries to achieve, is listed in Table 8.

Goal	STRIPS Goal States	Triggers
Initialise		
	IsAvailable(TTC_RX1)	IsOn(OBC) ^ Mode(Init)
	IsAvailable(TTC_TX1)	IsOn(OBC) ^ Mode(Init)
	IsAvailable(TTC_RX2)	IsOn(OBC) ^ Mode(Init)
	IsAvailable(TTC_TX2)	IsOn(OBC) ^ Mode(Init)
	IsAvailable(ADCS)	IsOn(OBC) ^ Mode(Init)
	IsAvailable(GPS_A)	IsOn(OBC) ^ Mode(Init)
	IsAvailable(GPS_P)	IsOn(OBC) ^ Mode(Init)
	IsAvailable(XBT1)	IsOn(OBC) ^ Mode(Init)
	IsAvailable(XBT2)	IsOn(OBC) ^ Mode(Init)
	IsAvailable(Iris)	IsOn(OBC) ^ Mode(Init)
	IsAvailable(ADAM)	IsOn(OBC) ^ Mode(Init)
	IsAvailable(PPU)	IsOn(OBC) ^ Mode(Init)
Initialise_ADCS		
	IsADCS_Mode(Sun_Pointing)	IsOn(OBC) ^Mode(Init)
	IsMode(Idle)	ADCS_Mode(Sun_Pointing)
GS_Downlink		(GSInRange TRUE)
	DownlinkOn)	
	RTTMEnabled	
	ActivityLogGS	
	FaultReportGS	
	MTDataGS	
	ImageDataGS	
Imaging_Op		
	ImageDataAq	*Trigger*

	Mode(Idle)	ImageDataAq
ADAM_Op		
	ADAMDataAq	*Trigger*
	Mode(Idle)	ADAMDataAq

Table 8 Goal macros and STRIPS goal sequence expansions

The planner takes only one goal at a time; once a plan for this goal is achieved, it goes back to the goal list and parses it for the next active goal.

The path from the ground goal macro to the actual execution thus involves a few layers of processing and decoding. First, the goal macro is decomposed into its corresponding STRIPS goal states. As each trigger condition for the STRIPS goal is met, a plan to achieve that state is generated. Each plan generated by the STRIPS based planner consists of a sequence of 'actions' that will bring about the desired changes to the system. Each action is mapped to a specific OBC execution command, which can be executed by the execution module in the baseline kernel.

The actions available to the OBC are listed in the Action Schema list. The action schema for each action specifies the preconditions and effects of the action. For scheduling and execution of time based goals, the mission planner must be able to have an estimate of the execution time of a command, ie, the time between the command being released for execution and the its corresponding effects becoming valid. In order to capture this information, the OBC action schema list will also have an associated timeout field for each action. This is a worst-case estimate of the execution time for the command. This entry is not shown in Table 9.

Action	Precondition	Add	Delete
On(x)	IsAvailable(x) ^PowerSuff(x) ^IsBus(x)	IsOn(x)	
Off(x)	IsBus(x)		IsOn(x)
OnPayload(x)	IsAvailable(x) ^PowerSuff(x) ^IsPayload(x) ^Mode(Idle)	IsOn(x) ^Mode(x)	Mode(Idle)
OffPayload(x)	IsPayload(x)		Mode(x)
Check(x)	IsOn(OBC) ^IsOn(x)	IsAvailable(x)	
CheckTM(t)	IsOn(OBC) ^IsOn(x)	IsOk(t)	
CheckLink(x)	IsOn(OBC) ^IsOn(x)	LinkAvail(x)	
SetADCS(m)		ADCS_Mode(m)	ADCS_Mode(m_{old})
PointADCS(att)		IsAttitude(att)	
Initialise(x)	IsOn(x) ^PowerSuff(x) ^LinkAvail(x)	IsReady(x)	
EnableRTTM	InRange(GroundStation)	RTTM_Enabled	
ToIdle	ADCS_Mode(Sun_Pointing) ^!IsOn(XBT) ^!IsOn(IRIS) ^!IsOn(ADAM) ^!IsOn(PPU)	Mode(Idle)	
ToMode(x)	Mode(Safe)	Mode(x)	Mode(Safe)
Dnld(x)	InRange(GroundStation)	Dnlded(x)	DataAcquired(x)
xbtDnld(x)	InRange(GroundStati	XbtDnlded(x)	ResultOf(Iris)

	on) ^IsOn(XBT) ^IsReady(XBT)		
OperateIris (loc,att)	IsOn(Iris) ^PowerSuff(Iris) ^IsReady(Iris) ^IsLocation (loc) ^IsAttitude (att) ^LinkAvail(Iris)	IRIS_ImageAcquired(loc,att)	IsLocation (loc) ^IsAttitude (att)
OperateADAM	IsOn(ADAM) ^PowerSuff(ADAM) ^IsReady(ADAM) ^LinkAvail(ADAM)	DataAcquired(ADAM_Data)	
OperatePPU	IsOn(PPU) ^PowerSuff(PPU) ^IsReady(PPU) ^LinkAvail(PPU)	DataAcquired(PPU_Data)	
End(x)	IsActive(x)	IsReady(x)	
ToTime (t)		IsTime(t)	IsTime(t_{old})
ToLocation(l)		IsLoc(l)	IsLoc(l_{old})
ToSafe		Mode(Safe)	Mode(m_{old})
Remove(x)			IsAvailable(x) ^LinkAvail(x)
Add(x)	IsOn(x)	IsAvailable(x) ^LinkAvail(x)	
SwitchCan(x)		LinkAvail(x)	

Table 9 Action Schema List

The planning operation begins when a new goal is received into the planning algorithm as input. This goal is placed into the initially empty goal stack. The planner retrieves the current state representation of the system into the currents state structure. The following steps are then followed.

1. Take the first (leftmost) literal that hasn't been established from the first goal in the goal stack and attempt to prove its truth from the current state.
   If it is True, start again at step 1. with the next literal in the goal.
   If all the literals are true, goto Step 6.
2. If the literal is not True, look through the action schema list for an action that contains this literal in its add list.
3. Select the first unexplored action found with the required literal in the add list. Push the instantiated action to the top of the plan action stack and its instantiated preconditions to the top of the goal action stack.
   If an action is found repeat from Step 1.
   If no further actions can be found at the current level with the required effects go to step 4.
4. If the plan action stack is not empty, pop the last added action from the plan action stack and the corresponding precondition from the goal stack.
   Start again from step 1.
   If the plan action stack is empty, goto Step 5.
5. Generate a trace of the planning activity and a goal not possible report.
6. If the goal stack has more than 1 entity, pop the topmost entry and start again at step 1 with the new topmost goal.
   If the goal stack has only 1 goal and it is the main goal goto step 7.
7. Retrieve current state representation and compare it with the one used to generate the plan (to check for changes and validity of the plan).
   If they are not consistent, clear the plan action stack and restart the procedure from step 1 with the main goal at the top of the stack. If the state representations are consistent the plan is ready to be passed for execution.

The preconditions for commands also include predicates relating to time and other scheduling information. In order to make these conditions meet during the planning operation, dummy action schemas are provided so that the planner can achieve these preconditions. During execution, these actions translate to NULL command and are ignored. Note that the mission planner never updates the actual state specification. At the

beginning of its operation, it retrieves the latest state model from the state update model, and works on its local copy.

The OBC mission planning activity is thus a regressive, depth-first approach to search for the plan. The planning operation thus generates a sequence of actions that can change the state from the currents state to the goal state. Each action corresponds to a command that can be executed by the command execution module. The plan therefore directly translates into an executable script for the required goal. Along with the plan, a trace report of the planning operation is also generated and added to the activity log to be used for diagnostics purposes, if required.

6.2.4 Plan Release

Once the planning algorithm has generated a plan to achieve the goal, it is ready to send each of the actions to the execution module. In order to schedule the command release correctly, the planner samples the state specification maintained periodically and releases each command as its preconditions are satisfied in the model. This sampling and checking operation is done at a frequency of once every second.

If the expected effects of a released command to enable the subsequent command are not visible until the 'timeout' period of the command, the command is resent. This is done three times, after which the fault detection and handling mechanism takes over.

6.3 State Maintenance Agent

The dynamic nature of the system and the environment makes it necessary to periodically update the state specification information in order to maintain the current version of the state as accurately as possible. This is done by sampling the collected telemetry and status information; and updating the state specification with this information.

6.3.1 Mapping Real World Information to Model

The telemetry and status information of the satellite consists of all the currents being consumed by the individual sub-systems (as measured by the respective current sensing circuits); temperatures measured at all the high power locations such as the s-band

downlink, X-band downlink; command reception acknowledges; and any other sub-system or operation specific information that is received by the system monitor module. Table 12 shows the mapping from predicates to telemetry and status values received on the OBC.

Predicate	Corresponding TM / Status or Information
IsOn(x)	Power for Sub-System 'x' is On
IsBus(x)	Database
IsPayload(x)	Database
IsAvailable(x)	Check(x) and CheckTM(x) action returns success
NotAvailable(x)	Check(x) and CheckTM(x) action does not return success
IsReady(x)	Acknowledge status for initialise command
PowerSuff(x)	Power > $P_{'x'}$ (power level needed for operation of 'x')
PowerInsuff(x)	Power < $P_{'x'}$ (power level needed for operation of 'x')
RTTMEnabled	Status from ground interface module
ImageDataAq FALSE	Status report on board from payload operations
ActivityLogDnlded	Status report on board from ground interface module
FaultReportsDnlded	Status report on board from ground interface module
ImageDataAq TRUE	Status report on board from payload operations
ImageDataDnlded	Status report on board from ground interface module
MTDataAq FALSE	Status report on board from payload operations
MTDataAq TRUE	Status report on board from payload operations
MTDataDnlded	Status report on board from ground interface module
FaultReports TRUE	Status report on board from activities log
FaultReports FALSE	Status report on board from activities log
GSInRange TRUE	ADCS determination telemetry or Ground interface module status
GSInRange FALSE	ADCS determination telemetry or Ground interface module status
DiskNotFull	RamDisk telemetry

IsTime(t)	From telemetry
Downlink On	Telemetry status of ground link transmitter
IsAcrive XBT	X-Band telemetry
IsLocation(l)	ADCS telemetry
IsMode(m)	Mission mode from operations and telemetry
IsADCSMode(m)	ADCS telemetry
InitialisationDone	Telemetry and status report from sub-systems

Table 10 Mapping of predicates to TM / Status inputs to OBC

This module is also responsible for detection of faults and triggering of the fault-handling operation. It also has to do a consistency check to ensure the integrity of the received data. This is done by verifying the collected telemetry information against their respective expected values.

6.3.2 The Verification Database

The state specification and expected effects from actions are composed of logical literals while telemetry information comes in the form of real world data, i.e., numbers, status bits etc. In order to monitor the telemetry, a database with real world values of ranges for the telemetry has to be maintained. This is done through a verification database maintained by this module.

The Verification Database contains the acceptable values for collected telemetry and is periodically updated based on the state specification and *expected* changes based on commands released for execution. This contains the telemetry values acceptable for the 'expected state' of the satellite.

The verification database is organised based on sub-system. Corresponding to each sub-system is a corresponding 'powered on status' entry and entries for current / temperature or other specific information available from telemetry. Each sub-system in the database consists of two types of parameters. The first set is discrete parameters. This consists of information relating to the operational state of the sub-system, digital status etc. which can have only discrete values. The other set consists of analog parameters. Corresponding

to each analog parameter are four entries. These correspond to the adaptive (dynamic) range for the parameter and the absolute range for the parameter.

The adaptive range for a sampled parameter is dynamically calculated. This is based on the averaged out range of the last *n* collected samples of the parameter and the characteristics of the telemetry point being monitored. A moving window (pre-decided by the ground) is provided for each telemetry parameter and this is shifted with the respect to the collected value every sample cycle. This is to take into consideration the effects of degradation of electronics over time and account for increase in drainage currents for the devices over time. Thus, a gradual increase to a higher value by a sampled parameter will not trigger a fault, but any sudden spikes in value to the same higher value will be flagged as an error.

The absolute maximum gives the range the device is not expected to exceed through the lifetime of the mission. The window is always maintained within this range. Table 13 shows the structure of the verification database capturing the current and temperature telemetry ranges for all the sub-systems on X-Sat.

$T_x(HI) = \min(\Sigma\ (0 \text{ to } n)T_x(t)/n,\ T_x(M))$

$T_x(LO) = \max(\Sigma\ (0 \text{ to } n)T_x(t)/n,\ T_x(m))$

n = 10; n shows depth of averaging and T is the telemetry point.

Sub-System	Power Switch	Current				Temp.				TM/Status msg and timeout
		M	m	H	L	M	m	H	L	
TTC_TX1	1 / 0	T_M	T_m	T_{HI}	T_{LO}	T_M	T_m	T_{HI}	T_{LO}	[Ack(cmd_1),10]
TTC_TX2	1 / 0	T_M	T_m	T_{HI}	T_{LO}	T_M	T_m	T_{HI}	T_{LO}	…
TTC_RX1	1 / 0	T_M	T_m	T_{HI}	T_{LO}	T_M	T_m	T_{HI}	T_{LO}	…
TTC_RX2	1 / 0	T_M	T_m	T_{HI}	T_{LO}	T_M	T_m	T_{HI}	T_{LO}	…
PSU	1 / 0	T_M	T_m	T_{HI}	T_{LO}	T_M	T_m	T_{HI}	T_{LO}	…
ADCS	1 / 0	T_M	T_m	T_{HI}	T_{LO}	T_M	T_m	T_{HI}	T_{LO}	…
XBT1	1 / 0	T_M	T_m	T_{HI}	T_{LO}	T_M	T_m	T_{HI}	T_{LO}	[P_Sw = 0, 2]
XBT2	1 / 0	T_M	T_m	T_{HI}	T_{LO}	T_M	T_m	T_{HI}	T_{LO}	…

IRIS	1 / 0	T_M	T_m	T_{HI}	T_{LO}	T_M	T_m	T_{HI}	T_{LO}	…
ADAM	1 / 0	T_M	T_m	T_{HI}	T_{LO}	T_M	T_m	T_{HI}	T_{LO}	…
PPU	1 / 0	T_M	T_m	T_{HI}	T_{LO}	T_M	T_m	T_{HI}	T_{LO}	…

Table 11Generic verification database for current / temperature telemetry and command effects

M, m, H and L for the Analog telemetry represent the absolute maximum, absolute minimum, adaptive high and adaptive low limits respectively; which are updated as described earlier. The values that change dynamically are shown in blue shade while the fixed values are shown in red. Current intake values are checked against these ranges only for sub-systems with a Power Switch = 1 state, otherwise it is expected to be 0. For parameters that can have unpredictable but valid values or which can have any value without affecting the system, don't care 'X' entries are used. The last column is a list of expected messages from the sub-system along with the related timeouts. This is updated form the Action Effects List discussed in the next section.

The fault detection mechanism flags a fault only if the time duration exceeds three times the expected timeout period of the effect. This is to give the planner the chance to re-issue the command. The fault handler is then triggered with all the information available regarding the fault.

6.3.3 Update of Verification Database

The operational state of each sub-system and digital telemetry entries in the database are updated based on the state model and expected changes through the commands released for execution. As mentioned earlier, the expected telemetry database is updated with respect to the current state specification. In order to update the expected telemetry database, without introducing an inconsistency between the model and the real world, the expected changes to the state are maintained in a separate list.

Whenever an action is released for execution, it is also sent to the model maintenance agent. The actions effect literals are added to the Action Effects List. The verification database is thus updated from the state specification and from the expected changes list. As mentioned in the earlier section, the timeout entry corresponding to actions are used to wait for the effects to appear, before flagging it as a deviation.

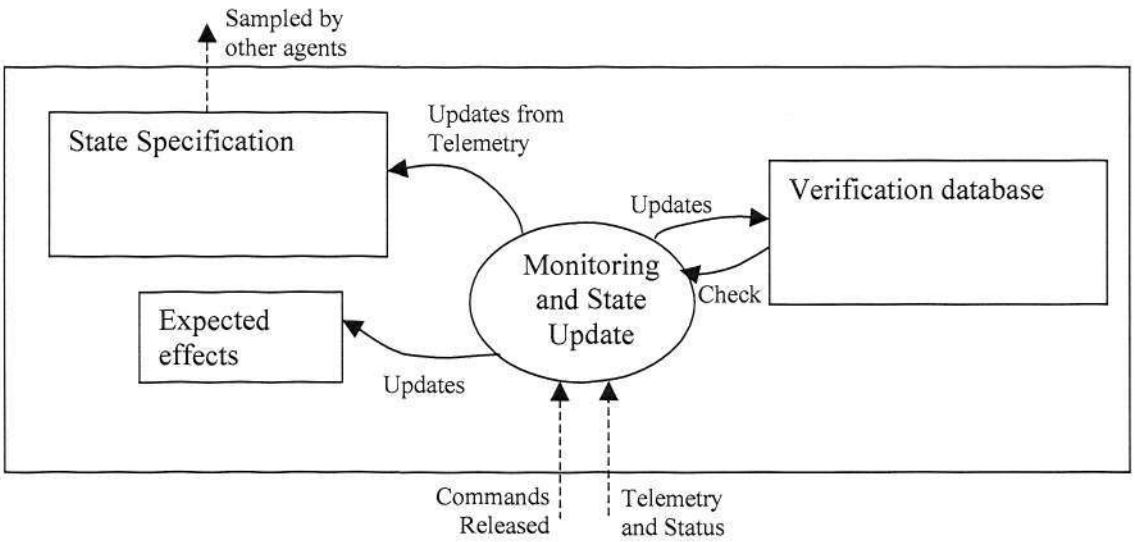

Table 12 State Monitoring and Specification Update

6.4 Fault Handler Agent

In a complex and error-prone environment, it is often difficult to infer the causes of anomalous observations in isolation. Knowledge of possible phenomena, their symptoms and observation of other environmental parameters enables a simple and robust method for diagnoses. This helps cross check anomalous values with the rest of the system rather than taking a decision based on a single anomaly. The internal operation of the fault handling module is intended to work in this manner and thus provide a high degree of fault detection and identification capability.

6.4.1 Overview

The fault handling operation on the OBC consists of four distinct phases. These are Detection, Classification, Action and Confirmation. The fault detection operation is done within the model update module and was discussed in the previous section. Once a fault is detected, the fault handling module is initiated with telemetry information regarding the fault. On detection of any fault, the plan execution is halted and toSafe command is released by the fault handler to go into Safe mode. During the fault handling operation no plan execution or mission planning activities are carried out. The fault handler saves the mode of the satellite prior to entering the Safe mode. At the end of a successful handling, this mode is restored.

Diagnosis of the fault characteristics and selection of what steps to take for recovery is classification. Once the fault is classified, the appropriate fault recovery actions are passed to the execution module for execution. This is the action phase. Once the selected action is taken, the state is monitored to confirm that the action does indeed result in the expected recovery. This is the confirmation of the fault handling. After a successful confirmation, the OBC comes out of Safe mode to the Normal mode of operation.

If the fault is unknown, or successful confirmation of the action is not achieved, a report of the fault including telemetry snapshot, expected state specification, action attempted and related specific information, is created for transmission to ground. The OBC then remains in Safe mode for future action from the ground.

The phases involved in classification and recovery from the fault are described here.

6.4.2 Classification

On fault detection, a snapshot set of the most recent telemetry values sampled, which deviates from the expected range, is passed to the fault handling module and it is initiated. The information passed to the fault handling module consists of the following information.

i. Telemetry snapshot
ii. List of telemetry values not correct
iii. Information about the deviation, (VERY HIGH / HIGH / LOW / ON / OFF)

The module also has read access to information in both the current state specification and the expected state database.

The fault handler is provided with a database of fault models of the satellite. The fault is classified depending on the fault model that corresponds to the snapshot and the appropriate handling action is taken.

The fault models database consists of an entry corresponding to each telemetry point on the satellite. The entries consist of level information (VERY HIGH, HIGH or LOW), status information (ON or OFF) or don't care entries (X) to indicate that the specific telemetry point is not relevant to the particular fault model.

The deviating set is compared against each of the entries in the database to identify the fault model entry that it matches the most. This correlation is carried out by comparing each telemetry entry in the fault model and the corresponding entry in the telemetry snapshot.

The faults can be classified based on their nature and type of handling required. The comparison with the fault models table results in the identification of the type of handling required for the fault. The faults can be grouped as:

i. Link fault: A fault on any of the internal communication links on the satellite, for which recovery action can be taken.
ii. Power fault: Any fault that is caused due to lack of sufficient power on the power supply.
iii. Sub-system power fault: Any fault that is caused due to higher than expected consumption of current or internal power problems within a sub-system.
iv. Bus sub-system functional fault: Fault caused due to a functional error on any of the bus sub-systems, i.e., ADCS, RamDisk, PSS, TTC, X-Band.
v. Payload functional fault: Fault caused due to a functional error on any of the payload sub-systems, i.e., IRIS, ADAM, PPU.
vi. Unknown fault: Unidentifiable fault.

The initial set of faults in the table consists of communication link failures and sub-system failures. Any fault detected that does not correspond to any of the faults, i.e., unknown fault, in the database is logged. The satellite is then set to SAFE mode and the report is sent to the ground. Once the ground diagnoses the fault and identifies a possible recovery operation, it can update the fault model database with the new entry and corresponding action.

Model #	Telemetry Model				Plan
1	$XBT1_{CURR}$ = HIGH	$XBT1_{TEMP}$ = HIGH	$XBT1_{SW}$ = 1	Others = OK	#1
2	$XBT1_{CURR}$ = HIGH	$XBT1_{SW}$ = 0		Others = OK	#2
…					

Table 13 Sample entries for the fault models database

The first version of the AFH software has very limited classification and identification capability. The capability of the fault classification database is expected to increase over time, both during fault simulation testing phase with the system and during the early phase of the mission. Ground station updates to the fault models and the recovery plans enable this.

6.4.3 Action

Corresponding to each entry in the fault models database is a recovery plan. The fault recovery operation is a pre-compiled series of actions or commands that is passed for execution. This is similar to the plans that are generated and passed for execution by the mission planner. However, the fault handling plans are not generated on the fly and are pre-determined sequences corresponding to fault model entries in the database.

When invoked, the fault handler module immediately sends a 'toSafe' action to the plan execution to be released for execution immediately. This ensures that the satellite immediately goes to a Safe mode and stays in that mode till fault handling is complete. After this, a comparison of the current state snapshot is done with each entry in the fault models database. If any of them matches the telemetry snapshot, the corresponding recovery plan is selected for execution. The plan release mechanism used is the same as the approach used by the mission planner.

Plan #1 $[10]_{TIMEOUT}$	Plan #2 $[2]_{TIMEOUT}$
Off(XBT1)	SET(Verification_database.XBT1.Temp = X)
Remove(XBT1)	
On(XBT2)	
Add(XBT2)	

Table 14 Sample fault action plans

Each action corresponds to either an OBC command or a configuration command for the verification database. Each recovery plan also includes a time entry in seconds, which is used by the confirmation phase to confirm that the recovery plan has been successful. The first version of the AFH software has a limited set of handling options. The set of recovery action plans is intended to be increased over time as the number of entries in the fault models database increases.

6.4.4 Confirmation

The confirmation phase consists of monitoring the state specification to ensure that the effects expected of the recovery plan have been achieved and are reflected in the

telemetry. The fault handling module does not actively involve in this. This is carried out by the state update module in the same way that it does monitoring of telemetry during normal satellite operation and plan execution. Any deviations from the expected state are detected and the fault handling module is notified.

The fault handling module waits for a pre-determined period of time (this is the entry in the recovery plan) to receive any fault detection notification from the state update module. After the timeout period, the mode is changed from Safe back to previous mode through the ToMode(x) action. If any notification is received from the state update, the OBC generates a fault report, including the state specification snapshot and verification database, for the ground to diagnose and remains in Safe mode.

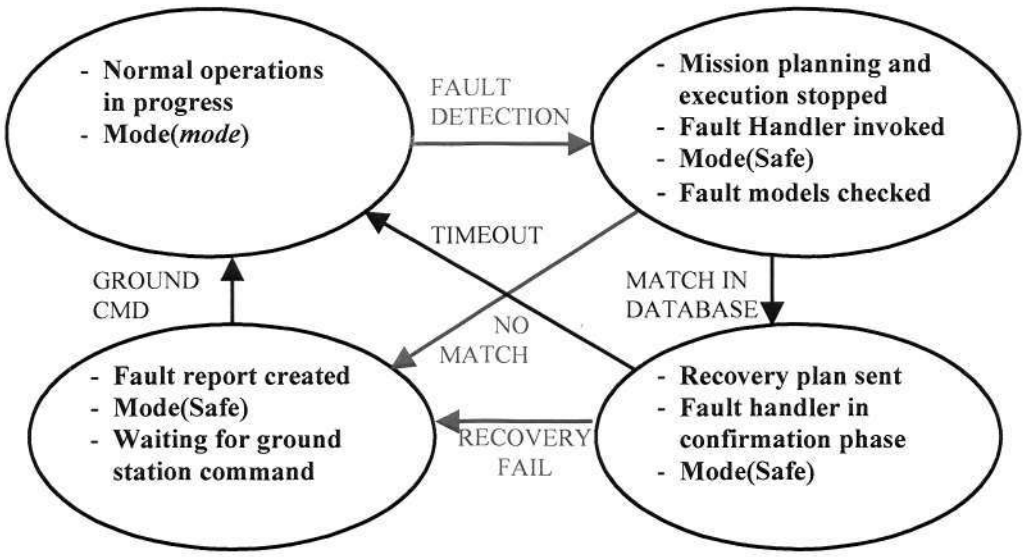

Figure 35 Fault Handling Operations

As mentioned earlier, the current design of the fault handler does not include an exhaustive list of fault models. An initial testing and commissioning phase with the satellite system would be required to build up a comprehensive database.

7. Conclusion

7.1 Results

7.1.1 Hardware Platform and Baseline Operations

A prototype of the fault tolerant platform and the baseline operations layer has been developed. This was functionally tested to validate operations. The system was also integrated with various other sub-system prototypes available and interface tests were carried out.

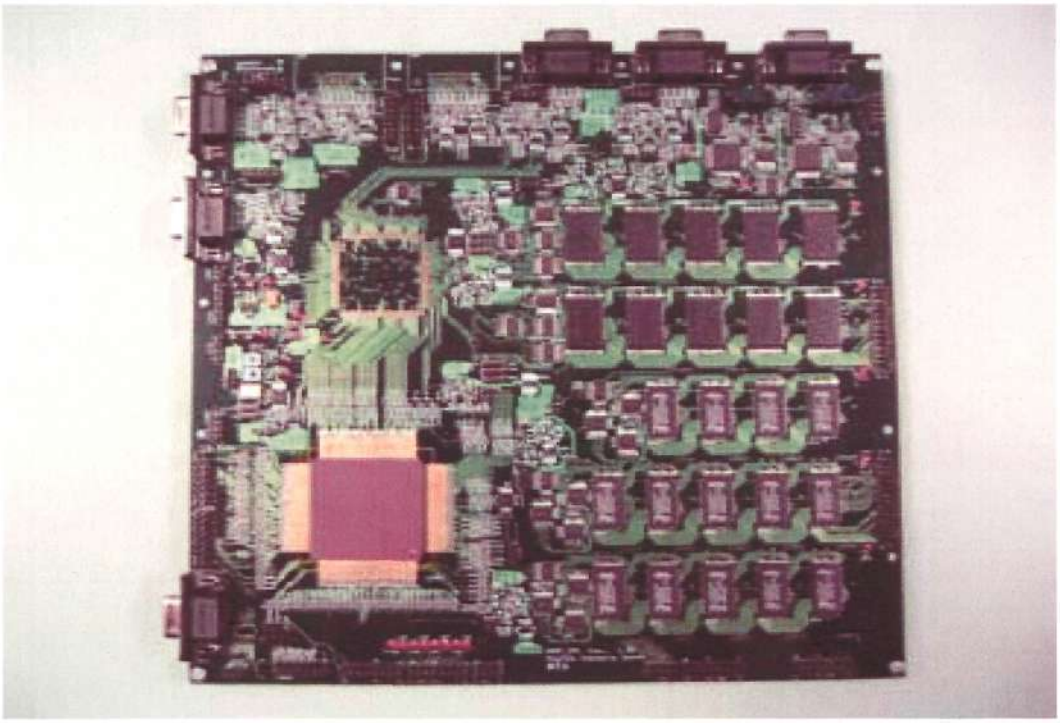

Figure 36 Prototype Platform of Flight Computer

This platform is based on the VxWorks RTOS. It supports the capabilities defined for the platform, including a boot loader, fault tolerance using dual redundancy, the peripheral device drivers for the CAN and a task manager to handle system initialisation, memory scrubbing and launching of the OBC processes (vxWorks tasks). These were discussed in Chapter 3 and 4.

The set of processes launched by the task manager are the baseline operations processes. These are the System Monitoring and Telemetry Collection process; the Command

Execution process and the Ground Interface process. The current Ground Interface module is a simulator and does not handle protocol details. These details are still under definition at the system level. Functioning versions of the telemetry collection process and the command execution process have been implemented. These have been successfully evaluated with sub-systems that have reached the prototype phase and are able to support interface testing.

Interface interactions for telemetry monitoring and command execution that have been carried out are X-Band telemetry collection and commanding; ADCS telemetry collection and commanding; PPU commanding; Accord GPS receiver; and sub-system power switching through Power Supply Unit commands. The rest of the sub-systems are expected to be tested once the interface and operations are fully defined and prototypes are available.

The baseline modules provide their interfaces over a debug console interface to the host PC. Command and response interactions with the baseline tasks are carried out over this link. This interface was used to provide the test commands for the interfaces and operations and receive responses.

7.1.2 Evaluation of Autonomous Mission Planner

A LISP based STRIPS planner was used to evaluate the planning approach and study it under various scenarios. The planner used was the IDM planner [25]. IDM uses a regressive planning approach to identify the action sequence required to achieve the goal. This is well suited for a microsatellite environment due to the limited number of action paths leading to the goal state. The satellite state specification and action schemas were implemented in LISP and the plans generated for autonomous activities were evaluated. The scenario used to evaluate the planner and the results are discussed below.

Autonomous Mission Profile

During the sunlit pass, the satellite is required to image over land bodies periodically. Each image is sent to the PPU for analysis. The image-processing algorithm in the PPU

determines if the area imaged is of interest to the mission or not. The PPU, in response to the command to analyse the image, analyses the image and sends a response indicating if the area is of interest or not.

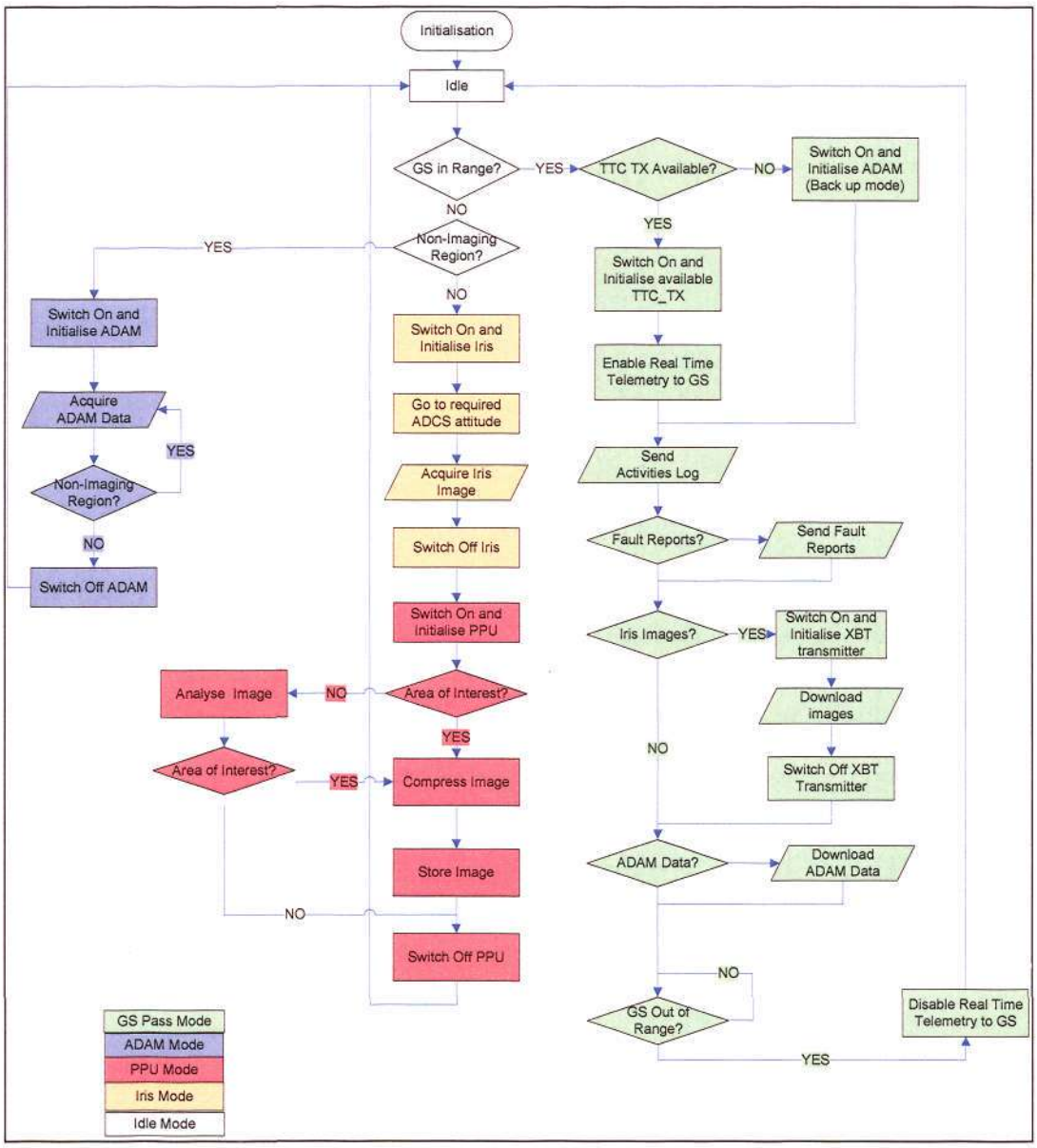

Figure 37 Flowchart of Activities for Autonomous Operation

Once an area of interest has been identified, the mission takes multiple images of the area of interest and surrounding areas, until the storage space on board is full. In order to

optimise usage of on-board storage space, an image compression algorithm PPU compresses each image that is acquired for storage. These images are then downloaded during the subsequent ground station contact.

When the satellite is over the ocean, the ADAM payload is operated to collect oceanographic data from mobile terminals. The acquired data is then downloaded during the ground station pass.

When the ground station comes into range of the satellite, the TT&C transmitter is switched on and real time telemetry transmission is initiated. An activities log of the mission and any fault information generated on board since the last contact are sent to the ground station over the TT&C link. Following this, all payload data is sent. If both primary and secondary TT&C links are unavailable, the radio transmitter on ADAM is used to try and downlink logged telemetry and payload data. All these activities are scheduled, subject to resource constraints, and carried out through commands from the flight computer.

The flowchart shows the sequence of activities that have to be managed by the flight computer for the mission profile described earlier. This flowchart does not show checks to confirm resource and power availability. In the Mission Planner, these are captured in terms of mission goals and triggers. These in turn are broken down to STRIPS planning goals at the time of planning. The tests done were at the STRIPS goals level to evaluate the computational complexity, efficiency and success-rate of the scheme.

Since STRIPS does not support the use of numbers in its planning and evaluation, numeric information such as power levels, time and location have been represented as literals. In the actual implementation these would be updated continuously, by the state maintenance agent, based on the system telemetry it receives.

Figure 39 shows the plans generated by the STRIPS planner under the various mission modes, put together in one plan action space. The 5 different shades signify the different

activities the actions are carrying out, corresponding to the flow chart in Figure 37. The plans are consistent with what is feasible.

The planning algorithm is fired by the mission goal and it generates the action sequence for the particular mission operation. The mission goal is selected based on the orbital position. This information is captured in the state specification through a database of propositional literals as described in Chapter 6. The different areas of interest for the mission are captured as literals in the database. The orbit location information, which is captured in the specification is periodically updated through telemetry and checked against these to identify the mission goal. For the simulated tests using LISP, a complete database was not developed; only a small database sufficient for the testing was created.

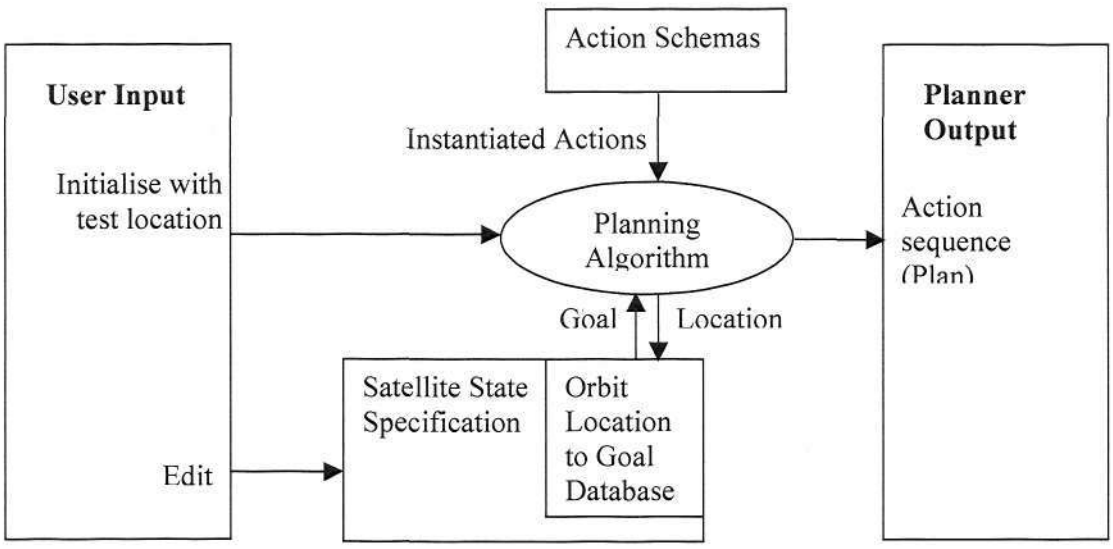

Figure 38 Block diagram of planner used for evaluation

The STRIPS planner generated these plans under different initial state and goal combinations. Some typical examples are captured in Table 15. The branching after some of the actions shows the different action paths that can be followed to achieve the goal. The planner chooses the appropriate path based on power considerations and sub-system availability.

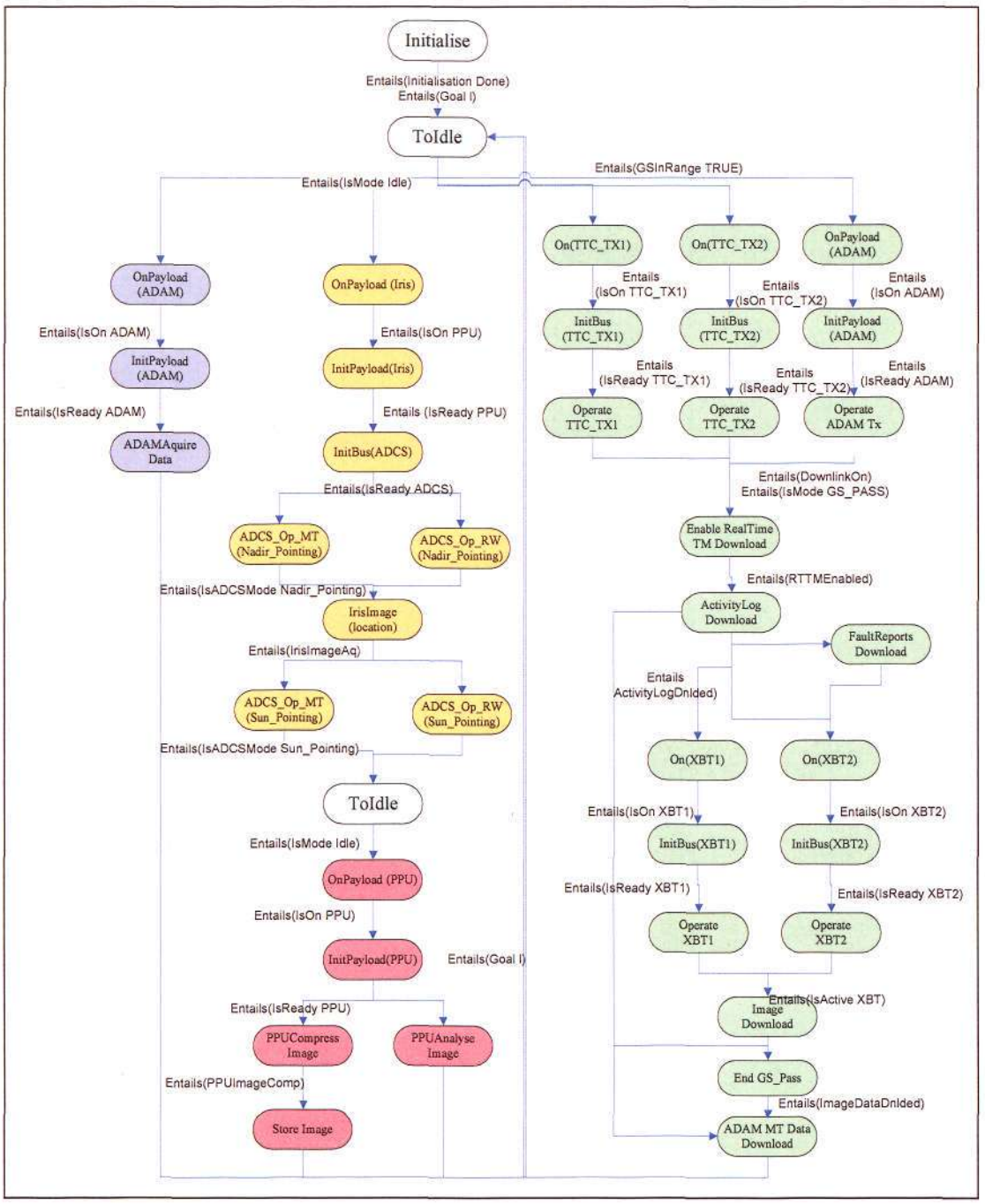

Figure 39 Plans generated by the mission planner for various mission goals

Although the system is currently running on a PC platform, an idea of the order of complexity involved in the planning can be obtained by the number of action schemas

expanded during the search for the plan. Table 15 shows the number of actions searched for some typical planning operations.

Initial State	Goals	Actions paths searched
Idle, All Systems OK, Power level high	Image area of interest, compress and store	39
Idle, All Systems OK, Power level high	Ground station pass operation	12
TT&C transmitter not available, Primary XBT transmitter not available	Ground station pass operation	12
Idle, All Systems OK, Power level high	ADAM Operation	4

Table 15 Action paths searched during planning

Table 15 however does not capture re-plan scenarios, where the initial state specifications are not valid at the end of the planning operation. In the actual satellite operation, the environment may change in this manner once in a while so these have to be considered while attempting to determine the complexity of the planner.

This system is currently running on a PC based LISP interpreter. Further initial state specifications and goal scenarios are being tested to study this and develop heuristics that would help reduce the search. Evaluation of the system in the embedded target will also help get a better estimate of the computational time taken for the planning operation.

7.2 Conclusion and Future Work

The research and development carried out for this project has established a framework and architecture that can be used for the implementation of an autonomous flight computer for LEO micro-satellites. The feasibility and advantages of this scheme have been evaluated and a working prototype implementing the hardware and baseline flight software has been developed in the laboratory. This has been designed targeting the X-

Sat micro satellite design and is currently being evaluated in the laboratory with other prototype sub-systems.

The mission planning operation and the state representation are being evaluated using a STRIPS based LISP planner. Typical scenario simulations and the plans generated are as expected. Further scenarios are being developed and would be tested in order to get a more comprehensive assessment.

What has been demonstrated in this thesis is an understanding of the issues involved in the building of micro-satellite platforms with autonomy features. A basic framework architecture to implement autonomy has been outlined. The system supports features for autonomous fault handling and mission related planning. A baseline version of this has been implemented demonstrating hardware fault tolerance and baseline flight software modules have been implemented targeting the X-Sat micro-satellite. The mission planner knowledge base has been implemented for a LISP based STRIPS planner and various scenarios have been tested with this.

A completely functioning version of the autonomous computer would need to have implementations of the agents needed for autonomy into the embedded software platform. This version would also necessarily support exhaustive fault handling capability.

Future work to be done on the system includes testing with actual prototypes of all sub-systems and the development, from testing and analysis, of a detailed database for faults and their classification. The fault classification and handling scheme needs to be tested through simulation and evaluated.

It would be useful to port the Mission Planner to the flight computer platform, run it and test it together with the baseline software. There are a number of approaches to implement this.

One possible way of doing this is to implement an interpreter to run on the embedded platform. This would be run as a task on the OS and accept LISP code and execute it. When triggered, this can be run with the current state model (initial state) and goal to generate a plan. Essentially, this would enable straightforward porting LISP code into the flight software. The interpreter would be a parser converting LISP functions and commands into C based executables. Based on an estimate of roughly 100 LISP primitives, and using function pointers and linked lists to simulate a LISP like operation, the interpreter software would be in the order of roughly 8000-10000 lines of code. This would involve a significant amount of development work intially, but once developed, it will enable the focus to be on the planning and the state specification instead of the implementation details. However, this may have a high amount of performance overhead. On a PC platform the interpreter works without any problems for even complex LISP functions, in an embedded platform running on a 10MHz processor, this scheme may overload the processor.

The alternative is to implement the algorithm and the structures using C language and run it directly as a task within the OS. This would enable higher performance and speed of execution. The algorithm, once developed and tested on a LISP platform, would be ported to native 'C' code for the VxWorks OS. This would enable a more efficient code for the planning operation and higher performance. In order to support testing of updated specification models and addition of new actions to the planner, the state specification literals and the action schemas that form the knowledge base would be captured in data structures. This would also enable dynamic upgrades to the system during operation. This option to integrate the planner with the baseline flight software will be explored further.

7.3 AI and Space

The work represented here on handling autonomy has exposed the potential that AI has for space missions. Undoubtedly space missions behaviourally can be much simpler than, for example, robots traversing difficult terrain. But there are complications which may not be in relation to behaviours but rather in ensuring the integrity of the platform in the harsh space environment.

This leads to a number of unexpected failures which cannot be anticipated. The failures could be single or potentially multiple, uncorrelated failures. Handling faults and carrying out repairs in such an environment is a challenge. Often the repair may not be complete and the platform has to be operated with severe constraints and limited resources. The challenge in space missions is to maximise the goal objectives that are reachable even under such circumstances.

The availability of high performance computing devices and the emergence of new software paradigms, both would motivate future space missions to be heavily driven by autonomy. Development of scalable computing systems to evaluate and validate autonomous operation is a step in this direction.

References

[1] *Small Satellites Home Page* http://centaur.sstl.co.uk/SSHP/

[2] *Surrey Satellite Technology Ltd* http://www.sstl.co.uk/index.php?loc=25

[3] *Smaller Satellites, Bigger Business : Concepts, Applications and Markets for Micro/Nanosatellites in a New Information World,* N. Crosby, Kluwer Academic Pub, January 2002.

[4] *Microsatellites and Nanosatellites, A brave new world*, Sir Martin Sweeting, Surrey Satellite Technology Limited, UK, 2001.

[5] *Evaluation of Radiation Effects in Flash Memories*, Tetsuo Miyahira and Gary Swift, Military and Aerospace Applications of Programmable Devices and Technologies Conference, September 1998.

[6] *The MIT Encyclopedia of the Cognitive Sciences* (MITECS)

[7] *Artificial Intelligence : A Modern Approach*, Stuart Russell and Peter Norvig

[8] *Computer Science as Empirical Enquiry: Symbols and Search*, Nevell and Simon

[9] *The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain*, Rosenblatt, 1958

[10] *Perceptrons*, Minsky and Papert, 1969

[11] *Perceptron Learning, MIT - Introduction to Neural Networks 2002*, Sebastian Seung

[12] *Understanding Intelligence*, Pfeifer and Scheier, 1999

[13] *How to build complete creatures rather than isolated cognitive simulators*, Brooks R. A.

[14] *A robust layered control system for a mobile robot*, Brooks, R.A., 1986

[15] *An Approach to Autonomous Operations for Remote Mobile Robotic Exploration* Caroline M. Chouinard, Forest Fisher, Daniel M. Gaines, Tara A. Estlin, Steve R. Schaffer, Jet Propulsion Laboratory, Proceedings of Aerospace Conference, 2003

[16] *The Techsat-21 Autonomous Space Science Agent*, Steve Chien, Rob Sherwood, Gregg Rabideau, Rebecca Castano, Ashley Davies, Michael Burl, Russell Knight, Tim Stough, Joe Roden (Jet Propulsion Laboratory) Paul Zetocha, Ross Wainwright, Pete Klupar, (Air Force Research Laboratory) Jim Van Gaasbeck,

Pat Cappelaere,Dean Oswald (Interface and Control Systems) , Proceedings of the first international joint conference on Autonomous agents and multiagent systems, 2002

[17] *The PROBA Onboard Autonomy Platform*, http://www.estec.esa.nl/proba/

[18] *A Model-based Approach to Reactive Self-Configuring Systems*, B.C. Williams and P.P. Nayak, Proceedings of the National Conference on Artificial Intelligence, 1996.

[19] *A Reactive Planner for a Model-based Executive*, Proceedings of International Joint Conference on Artificial Intelligence, 1997.

[20] *Autonomous Operations Through OnBoard Artificial Intelligence*, Rob Sherwood, Steve Chien, Rebecca Castano, Gregg Rabideau, Jet Propulsion Laboratory, International Conference on Space Operations, 2002

[21] *Autonomous Sequencing and Model-based Fault Protection for Space Interferometry*, Michel Ingham, Brian Williams (Massachusetts Institute of Technology), Thomas Lockhart, Amalaye Oyake, Micah Clark, Abdullah Aljabri (Jet Propulsion Laboratory), Proceedings of the 6 Symposium on Artificial Intelligence and Robotics & Automation in Space, 2001.

[22] *Exploring the Practical Limits of Operations Autonomy*, Glen E. Cameron, (The Johns Hopkins University Applied Physics Laboratory), Madeleine H. Marshall (Interface and Control Systems, Inc.), Proceedings of the Fifth SpaceOps Symposium, 1998

[23] *Casper: Space Exploration through Continuous Planning*, Russell Knight, Gregg Rabideau, Steve Chien, Barbara Engelhardt, and Rob Sherwood, Jet Propulsion Laboratory, IEEE Intelligent Systems, 2001

[24] *Space Radiation Effects on Electronic Components in Low-Earth Orbit*, NASA Preferred Reliability Practices

[25] *CMU Artificial Intelligence Repository*, http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/planning/systems/idm/0.html

[26] *An Autonomous Spacecraft Agent Prototype*, Barney Pell, Douglas E. Bernard, Steve A. Chien. Erann Gat, Nicola Muscettola, P. Pandurang Nayak, Michael D.

Wagner, Brian C. Williams, Proceedings of the first international conference on Autonomous agents, 1997

Appendix A – X-Sat Bus and Payloads

The X-Sat Mission

The X-Sat primary mission payload is an earth imaging system called IRIS developed as part of a commercial contract by Sarteci, South Korea. Secondary payloads are the Parallel Processing Unit (PPU) of NTU, which provides on-board image processing and classification capability; and the Advanced Data Acquisition and Messaging (ADAM) payload from Institute of Telecommunication Research in Australia, which is a communications payload designed to retrieve data from numerous sea-basedmobile terminals, in various parts of the world. Figure 1 gives a brief overview of the X-Sat bus and connectivity.

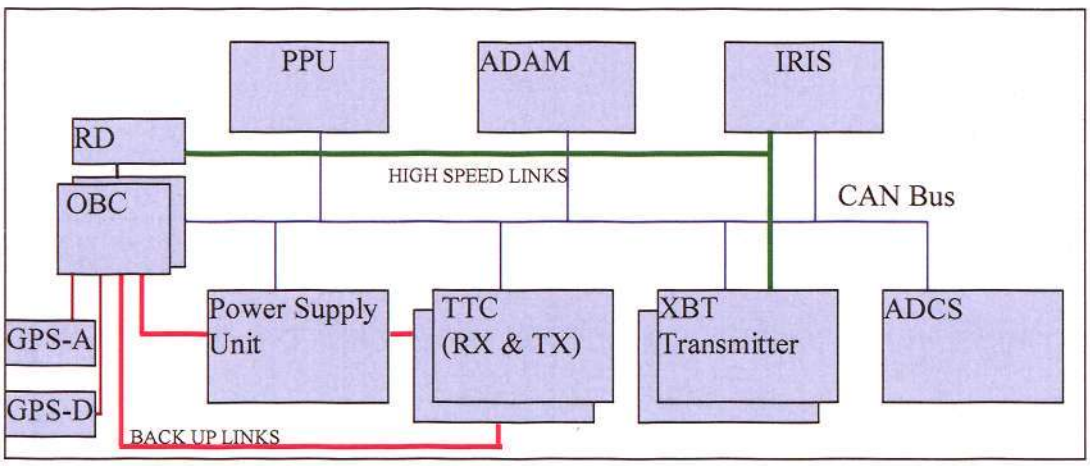

Figure 40 X-Sat Sub-Systems and Connectivity

Key Sub systems: The Telemetry, Tracking and Command unit (TT&C) is the ground interface over which the satellite receives commands and transmits telemetry data. The Power Supply Unit (PSU) handles power supply and switching. The Attitude Determination and Control System (ADCS) sub-system is responsible for attaining and maintaining required pointing state of X-Sat for various operations. The ADCS software runs on the ADCS processor, which is part of the ADCS sub-system. Dual redundant GPS receivers are used on board X-Sat for GPS position information. For high-speed downlink of large image data, there is an X-Band transmitter sub system. The Ram Disk (RD) is a mass storage module used to save mission data on X-Sat.

All the systems are operated through commands from the flight computer. The ground station controls and monitors the mission through command and telemetry interactions with the flight software. A cold redundant backup flight computer is available to cater for a failure in one of the flight computers.

Communications: The primary means of command dispatching and telemetry collection on X-Sat,spacecraft bus, is using the Controller Area Network (CAN) protocol. This protocol was originally developed for automotive and industrial applications; however, it has been used for several satellite missions including those by SSTL, University of Surrey Besides the CAN bus, there are dedicated back up links between critical systems (OBC-TTC-PSU) as well as high-speed links for image data transfers. The two GPS receivers also connect directly to the OBC. The dedicated links are shown in bold in Figure 1.

X-Sat Mission Modes: The X-Sat mission has a set of mission modes based on operations. The various mission operation modes and the activities corresponding to the mode are captured below.

i. IRIS Imaging: During imaging operation
    a. Check if power levels are sufficient for operation
    b. Turn on IRIS payload
    c. Configure RamDisk to receive IRIS data
    d. Command the ADCS to Nadir pointing
    e. Command IRIS to image (IRIS Image is sent to RamDisk over the high speed link)
    f. Turn on X-Band (high speed 50Mbit/s downlink)
    g. Download Image over X-Band to ground
    h. Turn off X-Band
    i. Turn off IRIS

ii. ADAM payload operation: During operation of the ADAM payload

a. Check is power levels are sufficient for operation
b. Turn on ADAM
c. Save data received through ADAM from ground terminals
d. Turn off ADAM

iii. PPU payload operation: During operation of the PPU payload
a. Check if power levels are sufficient for operation
b. Turn on the PPU
c. Send algorithm to PPU
d. Send image data to PPU
e. Receive processed data from PPU
f. Turn off PPU

iv. Ground station communication: During ground station pass and use of S-band TT&C sub-system for interactions with the ground
a. Send real time telemetry to the ground
b. Send WOD data to the ground
c. Send payload data to the ground (on request through commands)
d. Receive immediate and time scheduled commands and send them for execution

v. Safe: On detection of any non-correctable anomaly or failure on board
a. Put all sub-systems to Safe mode through commands
b. End all non-essential operations
c. Wait for ground command

vi. Idle: During the time the satellite is not in any of the above modes. Basic operations still maintained are:
a. System monitoring and telemetry collection
b. ADCS mode is maintained in Sun-pointing to enable charging of battery during sun-lit side pass.
c. Critical sub-systems such as PSU, TT&C Receivers, OBDH and ADCS are operating

Appendix B – Hardware Schematics

Appendix C –Mission Planner Knowledge Base (LISP)

```
;;;-----------------------------------------------------------------
;;;
;;;  XSat_STRIPS_MissionPlanner.lisp
;;;  Deepak Mohan
;;;  20 OCT 2005
;;;
;;;  First version of simple XSat state spec and image plan
;;;
;-----------------------------------------------------------
(reset-stripsops)
(setf *debug-strips* t)

;Goal actions
(strips-op Image_Compressed_Stored
                :filter          ( (OpIris ?l)
                                   (IsAOI ?l)
                        (PowerSuff Iris)
                        (PowerSuff PPU)
                        (DiskNotFull)  )
                :preconditions ( (IrisImageAq ?l)
                        (PPUImageComp)
                        (ImageStored)
                        (IsMode Idle))
                :add-list        ( (Goal ?l)
                                   (ImageDataAq TRUE)    )
                :delete-list     ( (IrisImageAq ?l)
                                   (PPUImageComp)
                                   (ImageDataAq FALSE)  )
                :variables       (?l)
                )

(strips-op Image_Location_Analysed
                :filter          ( (OpIris ?l)
                        (NotAOI ?l)
                        (PowerSuff Iris)
                        (PowerSuff PPU)
                        (DiskNotFull)  )
                :preconditions ( (IrisImageAq ?l)
                        (PPUImageAn)
                        (IsMode Idle))
                :add-list        ( (Goal ?l)
                                   (ImageDataAq TRUE)          )
                :delete-list     ( (IrisImageAq ?l)
                                   (PPUImageComp)
                                   (ImageDataAq FALSE)  )
                :variables       (?l)
```

```
                )

(strips-op Acquiring_MT_Data
          :filter       ((OpADAM ?l))
                :preconditions ((ADAMDataAq))
                :add-list       ((Goal ?l)
                                 (MTDataAq TRUE) )
                :delete-list    ( (ADAMDataAq)
                                  (MTDataAq FALSE)  )
                :variables      (?l)
                )

(strips-op End_GS_Pass
                :filter         ((IsMode GS_PASS)
                      (GSINRANGE FALSE))
                :preconditions ((IsMode Idle))
                :add-list       ((Goal ?l))
                :delete-list    ((RTTMEnabled))
                :variables      (?l)
                )

(strips-op GS_Downlink_Initialised
          :filter       ( (IsGS ?l)
                        (GSINRANGE TRUE) )
          :preconditions  ( (DownlinkOn)
                        (RTTMEnabled)
                        (ActivityLogGS)
                        (FaultReportGS)
                        (MTDataGS)
                        (ImageDataGS)   )
                :add-list       ( (Goal ?l) )
                :delete-list    ()
                :variables      (?l)
                )

;Need to see how post goal action is triggered!!!
;Post Goal Action
(strips-op toIdle
        :preconditions ((IsADCSMode Sun_Pointing)
                  (InitialisationDone))
        :add-list ((IsMode Idle))
        :delete-list ((IsMode Iris) (IsMode ADAM) (IsMode PPU)
                  (IsOn Iris) (IsReady Iris)
                  (IsOn ADAM) (IsReady ADAM)
                  (IsOn PPU) (IsReady PPU)  )
        :variables ()
```

```
        )

;Payload operations
(strips-op OnPayload
          :filter       ( (IsPayload ?p) )
          :preconditions  ( (IsMode Idle)
                            (IsAvailable ?p)
                    (PowerSuff ?p)     )
          :add-list      ( (IsOn ?p)
                            (IsMode ?p) )
          :delete-list    ( (IsMode Idle) )
          :variables      (?p)
          )

(strips-op InitPayload
          :filter       ( (IsPayload ?p) )
          :preconditions  ( (IsOn ?p)
                           (PowerSuff ?p) )
          :add-list      ( (IsReady ?p)  )
          :delete-list    ( )
          :variables      (?p)
          )

(strips-op Iris_Image_Loc
          :preconditions  ( (IsMode Iris)
                    (IsOn Iris)
                    (IsReady Iris)
                    (PowerSuff Iris)
                    (IsLocation ?l)
                    (IsADCSMode Nadir_Pointing))
          :add-list      ( (IrisImageAq ?l)
                    (NewImage)     )
          :delete-list    ( (IsLocation ?l) )
        :variables      (?l)
          )

(strips-op PPU_Compress_Image
                :filter           ( )
                :preconditions ((IsMode PPU)
                                (IsReady PPU)
                          (PowerSuff PPU)
                        (NewImage) )
                :add-list         ((PPUImageComp))
                :delete-list      ((NewImage))
                :variables        ()
                )
```

```
(strips-op PPU_Analyse_Image
            :filter          ( )
            :preconditions ((IsMode PPU)
                            (IsReady PPU)
                      (PowerSuff PPU)
                    (NewImage)     )
            :add-list        ((PPUImageAn))
        :delete-list  ((NewImage))
            :variables       ()
            )

(strips-op ADAM_Acquire_Data
        :filter      ()
            :preconditions ((IsMode ADAM)
                            (IsReady ADAM)
                      (PowerSuff ADAM))
            :add-list        ((ADAMDataAq))
            :delete-list     ()
            :variables       ()
            )

;Bus sub-system operations
(strips-op On
        :filter      ( (IsBus ?p) )
        :preconditions  ( (IsAvailable ?p)
                  (PowerSuff ?p)
                )
        :add-list     ( (IsOn ?p) )
        :delete-list   ( )
        :variables     (?p)
        )

(strips-op InitBus
        :filter      ( (IsBus ?p) )
        :preconditions  ( (IsOn ?p)
                  (PowerSuff ?p) )
        :add-list     ( (IsReady ?p) )
        :delete-list   ( )
        :variables     (?p)
        )

(strips-op Store_Image
      :filter      ( )
        :preconditions  ((PPUImageComp))
        :add-list     ( (ImageStored) )
```

```
        :delete-list   ( )
        :variables     ()
        )

(strips-op ADCS_Op_MTq
        :preconditions  ( (IsOn ADCS)
                (IsReady ADCS)
                (PowerSuff MTq)
                (PowerInsuff RW) )
      :add-list      ( (IsADCSMode new) )
        :delete-list   ( (IsADCSMode prev) )
        :variables     (?prev ?new)
        )

(strips-op ADCS_Op_RW
        :preconditions  ( (IsOn ADCS)
                (IsReady ADCS)
                (PowerSuff RW) )
      :add-list      ( (IsADCSMode ?new) )
        :delete-list   ( (IsADCSMode ?prev) )
        :variables     (?prev ?new)
        )

(strips-op toLocation       ; Dummy command
        :preconditions  ( )
        :add-list      ( (IsLocation ?l) )
        :delete-list   ( )
        :variables     (?l)
        )

(strips-op Operate_XBT1
        :filter   ( (IsAvailable XBT1)
                        (PowerSuff XBT1) )
        :preconditions  ( (IsOn XBT1)
                        (IsReady XBT1)
                        (PowerSuff XBT1)
                (DownlinkOn))
        :add-list      ( (IsActive XBT) )
        :delete-list   ( )
        :variables     ()
        )

(strips-op Operate_XBT2
        :filter   ( (IsAvailable XBT2)
                        (PowerSuff XBT2) )
        :preconditions  ( (IsOn XBT2)
```

```
                    (IsReady XBT2)
                    (PowerSuff XBT2)
                (DownlinkOn))
        :add-list     ( (IsActive XBT) )
        :delete-list  ( )
        :variables    ()
        )

(strips-op Operate_TTC_TX1
        :filter   ( (IsAvailable TTC_TX1) )
        :preconditions  ( (IsOn TTC_TX1)
                    (IsReady TTC_TX1)
                    (PowerSuff TTC_TX1) )
        :add-list     ( (DownlinkOn)
                    (IsMode GS_PASS) )
        :delete-list  ( (IsMode Idle) )
        :variables    ()
        )

(strips-op Operate_TTC_TX2
        :filter   ( (IsAvailable TTC_TX2) )
        :preconditions  ( (IsOn TTC_TX2)
                    (IsReady TTC_TX2)
                    (PowerSuff TTC_TX2)  )
        :add-list     ( (DownlinkOn)
                    (IsMode GS_PASS) )
        :delete-list  ( (IsMode Idle) )
        :variables    ()
        )

(strips-op Operate_ADAM_TX
            :filter   ((UnAvailable TTC_TX1)
                (UnAvailable TTC_TX2)
                (IsAvailable ADAM) )
            :preconditions  ( (IsOn ADAM)
                        (IsReady ADAM)
                        (PowerSuff ADAM)  )
            :add-list     ( (DownlinkOn)
                        (IsMode GS_PASS) )
            :delete-list  ( (IsMode Idle) )
            :variables    ()
        )

(strips-op Enable_RealTime_TM
        :preconditions  ( (DownlinkOn)
                    (IsMode GS_PASS) )
```

```
        :add-list     ( (RTTMEnabled) )
        :delete-list  ( )
        :variables    ()
        )

(strips-op ActivityLog_Dnld
        :preconditions  ( (DownlinkOn)
                          (IsMode GS_PASS) )
        :add-list     ( (ActivityLogGS) )
        :delete-list  ( )
        :variables    ()
        )

(strips-op FaultReportDnld
        :filter   ( (FaultReports TRUE) )
        :preconditions  ( (DownlinkOn)
                          (IsMode GS_PASS) )
        :add-list     ( (FaultReports FALSE)
                        (FaultReportGS) )
        :delete-list  ((FaultReports TRUE) )
        :variables    ( )
        )

(strips-op NoFaultReportDnld
        :filter   ( (FaultReports FALSE) )
        :preconditions  ( (DownlinkOn)
                          (IsMode GS_PASS) )
        :add-list     ( (FaultReportGS) )
        :delete-list  ( )
        :variables    ( )
        )


(strips-op ADAMDataDnld
        :filter   ( (MTDataAq TRUE) )
        :preconditions  ( (DownlinkOn)
                          (IsMode GS_PASS)
                (FaultReportGS)
                (ActivityLogGS) )
        :add-list     ( (MTDataGS)
                        (MTDataAq FALSE) )
        :delete-list  ((MTDataAq TRUE) )
        :variables    ( )
        )

(strips-op NoADAMDataDnld
```

```
        :filter   ( (MTDataAq FALSE) )
        :preconditions  ( (DownlinkOn)
                          (IsMode GS_PASS)
                    (FaultReportGS)
                    (ActivityLogGS) )
        :add-list       ( (MTDataGS) )
        :delete-list    ( )
        :variables      ( )
        )


(strips-op ImageDataDnld
        :filter   ( (ImageDataAq TRUE) )
        :preconditions  ( (IsMode GS_PASS)
                          (IsActive XBT)
                    (FaultReportGS)
                    (ActivityLogGS) )
        :add-list       ( (ImageDataGS)
                          (ImageDataAq FALSE) )
        :delete-list    ((ImageDataAq TRUE) )
        :variables      ( )
        )

(strips-op NoImageDataDnld
        :filter   ( (ImageDataAq FALSE) )
        :preconditions  ( (IsMode GS_PASS)
                    (FaultReportGS)
                    (ActivityLogGS) )
        :add-list       ( (ImageDataGS) )
        :delete-list    ( )
        :variables      ( )
        )

(strips-op PostPoneImageDnld
        :filter   ( (PowerInsuff XBT1)
                          (PowerInsuff XBT2))
        :preconditions  ( (IsMode GS_PASS) )
        :add-list       ( (ImageDataGS) )
        :delete-list    ( )
        :variables      ( )
        )

;Initial State

(setq Initial_State
  '(
```

```
;System Info
   (IsPayload Iris)
   (IsPayload PPU)
   (IsPayload ADAM)
   (IsBus ADCS)
   (IsBus TTC_TX1)
   (IsBus TTC_TX2)
   (IsBus XBT1)
   (IsBus XBT2)
   (IsBus PSU)

;System State
   (IsOn OBC)
   (IsOn ADCS)
   (IsAvailable PSU)
   (IsAvailable TTC_TX1)
   (IsAvailable TTC_TX2)
   (UnAvailable TTC_TX1)
   (UnAvailable TTC_TX2)
;  (IsAvailable XBT1)
   (IsAvailable XBT2)
   (IsAvailable ADCS)
   (IsAvailable Iris)
   (IsAvailable ADAM)
   (IsAvailable PPU)
   (IsAvailable ADCS)
   (IsAvailable CAN1)
   (IsAvailable CAN2)
   (IsAvailable BKUP)
   (IsMode GS_PASS)
   (DiskNotFull)
   (InitialisationDone)
;GS Data
   (ImageDataAq TRUE)
   (MTDataAq TRUE)
   (FaultReports TRUE)
;  (GSINRANGE TRUE)
   (GSINRANGE FALSE)

;Mission Info
   (OpIris loc1)
   (NotAOI loc1)
   (OpIris loc2)
   (IsAOI loc2)
   (OpAdam loc3)
   (IsGS loc4)
```

```
;Communication links


;Power Levels
;   (PowerSuff TTC_TX1)
;   (PowerSuff TTC_TX2)
    (PowerSuff XBT1)
    (PowerSuff XBT2)
    (PowerSuff ADCS)
    (PowerSuff MTq)
    (PowerSuff RW)
    (PowerSuff Iris)
    (PowerSuff PPU)
    (PowerSuff ADAM)
    ))

(defun testAn ()
  (strips
   Initial_State
   '((PPUImageAn loc)) ))

(defun testImage ()
  (strips
   Initial_State
   '((IrisImageAq loc)) ))

(defun test (var)
  (strips
   Initial_State
   var))

(defun autotestImageNonAOI ()
  (strips Initial_State '((Goal loc1))))

(defun autotestImageAOI ()
  (strips Initial_State '((Goal loc2))))

(defun autotestADAM ()
  (strips Initial_State '((Goal loc3))))

(defun autotestGSInit ()
  (strips Initial_State '((Goal loc4))))

(defun autotestGSDone ()
  (strips Initial_State '((Goal loc5))))
```