A Software Architecture for Autonomous Spacecraft

by

Jimmy S. Shih

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 1, 1997

Author ______________________
Department of Electrical Engineering and Computer Science
May 1, 1997

Certified by [illegible]
Patrick H. Winston
Thesis Supervisor

Certified by [illegible]
Arvind
Thesis Supervisor

Accepted by [illegible]
F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses

A Software Architecture for Autonomous Spacecraft

by

Jimmy S. Shih

Submitted to the

Department of Electrical Engineering and Computer Science

May 1, 1997

In Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The thesis objective is to design an autonomous spacecraft architecture to perform both deliberative and reactive behaviors. The Autonomous Small Planet In-situ Reaction to Events (ASPIRE) project uses this architecture to integrate several autonomous technologies for a comet orbiter mission. This architecture uses the deliberative path for performing deliberative behaviors, and the three bypass paths for performing reactive behaviors. The deliberative path subsumes the three bypass paths when it has time to handle events. The three bypass paths are used to provide faster response time. The ASPIRE project shows that the deliberative path can handle all the deliberative behaviors required by the mission, although, the three bypass paths for handling reactive behaviors are not implemented in the project. Finally, this thesis describes several good design and implementation elements.

Thesis Supervisor: Patrick H. Winston
Title: Professor, Director of the Artificial Intelligence Laboratory
Thesis Supervisor: Arvind
Title: Charles W & Jennifer C Johnson Professor

Table of Content

List of Figures

Preface and Acknowledgments

I started my research in autonomous spacecraft architecture when I worked on the Autonomous Serendipitous Science Acquisition for Planets (ASSAP) project at the Jet Propulsion Laboratory (JPL) during the summer of 1995. The ASSAP's goal was to integrate autonomous technologies for missions to map large planets. I came back to JPL the following year to work on the Autonomous Small Planet In-situ Reaction to Events (ASPIRE) project, the follow-on project to ASSAP. The ASPIRE's goal was to integrate autonomous technologies for missions to orbit comets. This thesis is based on the software architecture I designed for the ASPIRE project.

I want to thank Abdullah Aljabri, the task lead for the ASSAP project, for getting me started in research on autonomous spacecraft architecture. I want to thank Tooraj Kia, the task lead for the ASPIRE project, for giving me the chance to design ASPIRE's architecture. I want to thank members of the ASSAP and ASPIRE teams for their valuable advice. Finally, I want to thank my advisors, Professor Patrick Winston and Professor Arvind, for supervising my thesis.

List of Abbreviations

AFAST	Autonomous Feature and Star Tracker
ASPIRE	Autonomous Small Planet In-situ Reaction to Events
ASSAP	Autonomous Serendipitous Science Acquisition for Planets
DS1	Deep Space 1
JPL	Jet Propulsion Laboratory
MIR	Mode Identification and Recovery
RAPs	Reactive Action Packages
TCA	Task Control Architecture

1 ASPIRE Project

1.1 Thesis Objective

The thesis objective is to design a spacecraft software architecture for handling both deliberative and reactive behaviors. Autonomous spacecraft can use deliberative behaviors to achieve mission objectives and reactive behaviors to deal with unexpected events. The software architecture contains certain key ideas for controlling future spacecraft.

Autonomous spacecraft can further our ability to explore space by reducing spacecraft operation cost and by improving spacecraft response time. Autonomous spacecraft can reduce operation cost by automating deliberative behaviors onboard the spacecraft. Currently, we need many people and heavy usage of the Deep Space Network antennas to command spacecraft from the ground. Similarly, autonomous spacecraft can improve response time by conducting reactive behaviors onboard the spacecraft. Currently, we need telemetry linkup and long response time to command spacecraft from the ground. By reducing operation cost and improving response time, we can maintain more spacecraft and capture more scientific events.

1.2 ASPIRE Project Goal

The Autonomous Small Planets In-situ Reaction to Events (ASPIRE) project goal is to integrate new technology modules, such as the science, tracking, planner, and navigation modules, to support the quick reaction mode of a comet orbiter mission. In the quick reaction mode, the science module detects and identifies interesting targets. The tracking module detects and tracks ejected cometary fragments. The planner module generates plans for the spacecraft in response to the science generated targets. Finally, the navigation module produces plans to perform close flyby maneuvers around the comet. The ASPIRE project demonstrates in situ science gathering triggered by several unpredictable environmental changes in the comet.

The ASPIRE project demonstrates four new autonomous spacecraft concepts. First, the project demonstrates that the ASPIRE architecture framework can be used to integrate all the autonomy technologies. Second, it demonstrates that the science module's change detection algorithm can detect sub-pixel level movements on the comet (Crippen 1992). Third, it demonstrates that the tracking module's Autonomous Feature and Star Tracker (AFAST) algorithm can detect and track ejected cometary fragments (Chu et al. 1994). Fourth, it demonstrates that the navigation module's close proximity algorithm can perform close flyby maneuvers around the comet (Scheeres 1996). These four technologies are essential for spacecraft that want to orbit around the comet.

1.3 Mission Scenario

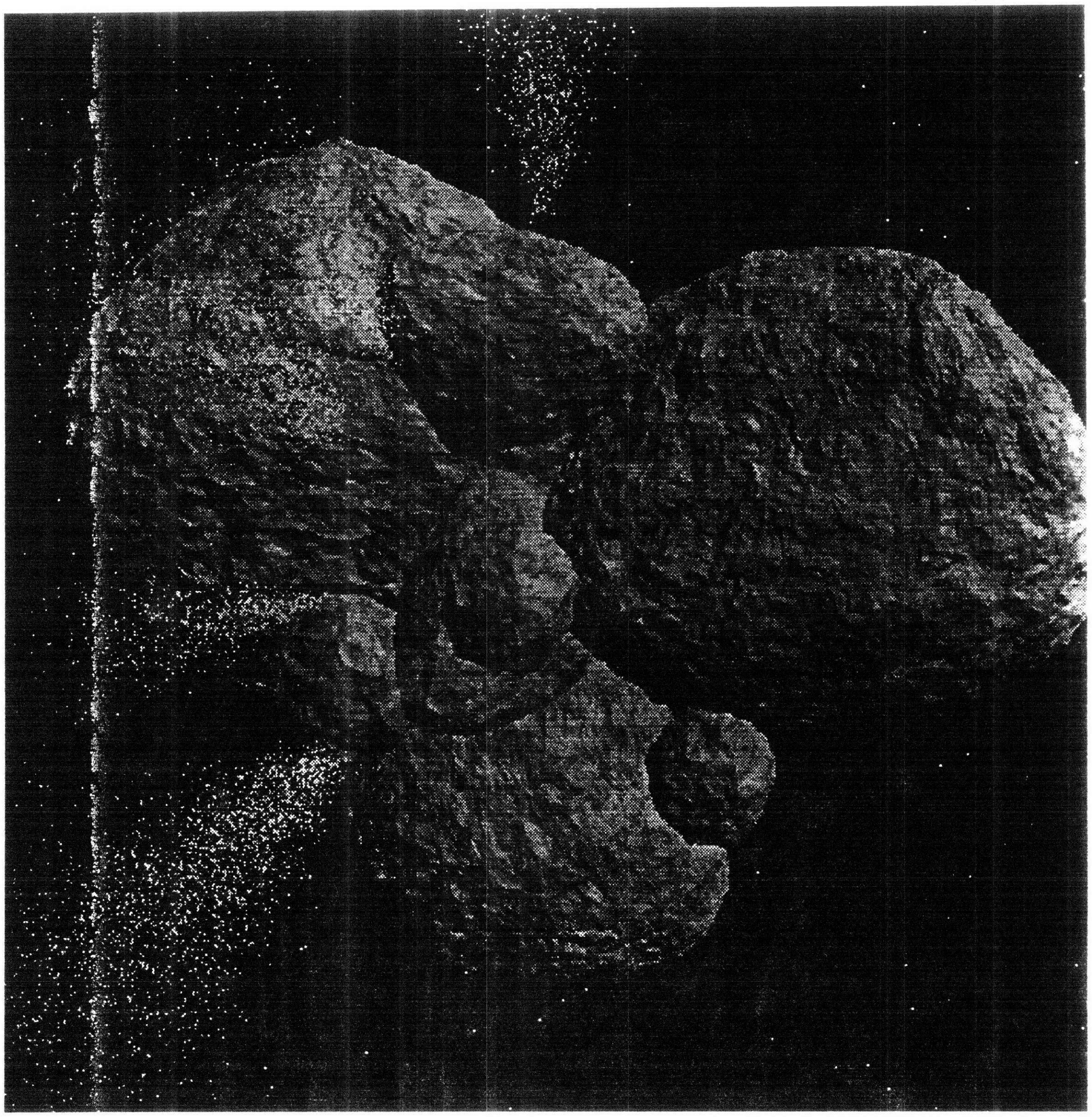

Figure 1 Comet Nucleus (created by William Lincoln 1996).

The ASPIRE project uses a comet model and a spacecraft model to simulate the environment for the mission. The comet model, shown in figure 1, is four kilometers wide in diameter. We think a comet is made up of many loosely held cometary fragments that are remnants of the early solar system building blocks. Three types of events can happen on a comet: cracks, ejected fragments, and comet-splitting. Cracks occur when two cometary fragments move against each other on the comet. Ejected fragments occur when cometary fragments are ejected from the comet. Comet-splitting occurs when the comet breaks apart into several large pieces. The spacecraft model contains two cameras to observe the comet. The wide-field camera always points at the comet center. The narrow-field camera can be controlled to take fine resolution pictures and track ejected fragments. The comet model and the spacecraft model are used to demonstrate the onboard autonomy software.

The mission is to observe a near inactive comet. The spacecraft uses the wide-field camera to take pictures of the comet upon arrival. These pictures are processed onboard the spacecraft to construct the internal comet model. After the spacecraft has generated the internal comet model, it enters the quick reaction mode. The spacecraft initially enters an orbit, 20 kilometers from the comet, to take wide-field images. The science module receives these wide-field images and compares them with images taken in the past to detect sub-pixel level and pixel level movements such as cracks and ejected fragments. The science module informs the planner module when it discovers interesting targets on the comet surface. The planner will then direct the tracking module to use the narrow-field camera to track them. The planner module also decides over which targets a close flyby maneuver shall be performed. A close flyby maneuver can place the spacecraft within 500 meters of the comet. After a close flyby maneuver, the spacecraft returns to an orbit around the comet to take more pictures and wait for the next close flyby command. When unexpected events happen, such as a comet-splitting, the spacecraft slowly maneuvers away to a safe distance of 100 kilometers from the comet to observe the events. The mission scenario tests all the autonomy algorithms by injecting various changes to the comet model at various times.

1.4 Road Map

This thesis is organized into five chapters and one appendix. Chapter one describes the ASPIRE project. Chapter two provides background literature on autonomous robot and autonomous spacecraft architectures. Chapter three describes the ASPIRE architecture framework. Chapter four describes the ASPIRE architecture implementation. Finally, Chapter five concludes with some recommendations. Appendix A provides interface description of the ASPIRE architecture implementation.

2 Background Literature

2.1 Autonomous Robot Architectures

Autonomous robots and autonomous spacecraft are similar in many respects. Both need to perform complex tasks in uncertain environment with limited sensors, actuators, and power. Furthermore, both need to degrade gracefully when faults occur. Therefore, many good ideas use in designing architectures for autonomous robots can be used in designing architectures for autonomous spacecraft.

However, autonomous robots are different from autonomous spacecraft in four areas. First, autonomous robots are more self-sufficient than autonomous spacecraft. Autonomous robots are designed to perform tasks without additional support once they are placed in the environment. Autonomous spacecraft, on the other hand, are designed mainly to reduce ground control. Second, autonomous robots are less capable of interpreting sensor data than autonomous spacecraft (Pell et al. 1996). A zero reading from a robot's ground sensor can mean that the robot is approaching a hole, a slope, or a cliff. But, a measurement from a spacecraft's Inertial Reference Unit can determine spacecraft's exact attitude. Third, autonomous robots are better at handling faults than autonomous spacecraft (Pell et al. 1996). Autonomous robots can react to a fault by trying different actions without knowing the cause of the fault. Autonomous spacecraft, on the other hand, must view each fault as a failure, and must act according to the context and result of the fault. Fourth, autonomous robots are more prone to suffer transient failures than autonomous spacecraft (Pell et al. 1996). Failures occurring in autonomous robots are more likely to be temporary. In contrast, failures occurring in autonomous spacecraft are more likely to be permanent. Even though autonomous robots and autonomous spacecraft are similar in many areas, they do have several subtle differences.

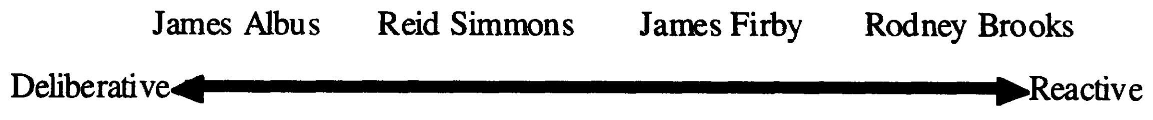

Figure 2 Four Distinct Autonomous Robot Architectures.

Four people, James Albus, Reid Simmons, James Firby, and Rodney Brooks, have proposed four distinct architectures for controlling autonomous robots. Their architectures, shown in figure 2, differ in their emphases on performing deliberative and reactive behaviors. James Albus's architecture emphasizes performing deliberative behaviors. Reid Simmons's architecture emphasizes performing deliberative behaviors, but contains mechanisms for performing reactive behaviors. In contrast, James Firby's architecture emphasizes performing reactive behaviors, but contains mechanisms for performing deliberative behaviors. Finally, Rodney Brooks's architecture emphasizes mainly on performing reactive behaviors. These four architectures provide four distinct philosophies for controlling autonomous robots.

2.1.1 James Albus's Architecture

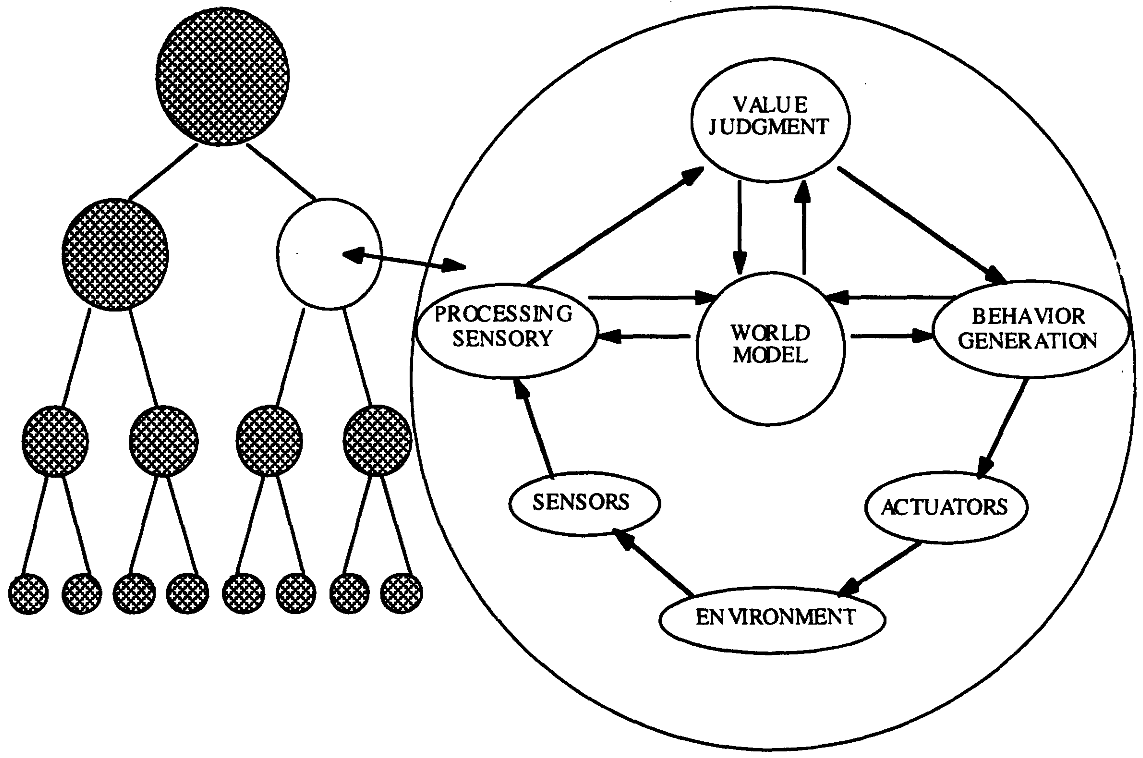

Figure 3 James Albus's Architecture (1991).

James Albus proposes an architecture for autonomous robots that emphasizes performing deliberative behaviors. The architecture, shown in the left side of figure 3, is a hierarchy. From one hierarchy level down to the next lower hierarchy level, the time duration of interests for perception resolution, world modeling, goal planning, and control bandwidth decreases by an order of magnitude. For example, the highest level of the hierarchy only deals with monthly events. The next lower level only deals with daily events. And the next lower level only deals with hourly events and so forth. The architecture, shown in the right side of figure 3, uses six basic modules, Sensors, Sensory-Processing, World-Model, Value-Judgment, Behavior-Generation, and Actuators, to perform deliberative behaviors at each hierarchy level. The Sensor-Processing module receives inputs about the Environment from the Sensors module. Information from the Sensor-Processing module are then stored in the World-Model module. The Value-Judgment module uses information in the World-Model module to issue appropriate high level commands. The Behavior-Generation module then decomposes high level commands from the Value-Judgment module into low level commands, and executes low level commands on the Environment using the Actuators module. The architecture only needs to perform deliberative behaviors because lower and lower levels of the hierarchy provide faster and faster response time for handling unexpected events (Albus 1991).

James Albus's reason for designing an architecture that emphasizes performing deliberative behaviors is because he believes that deliberative behaviors can bring about learning better than reactive behaviors can. An architecture that emphasizes deliberative behaviors can propagate new knowledge globally to all other modules through the World-Model module. But an architecture that emphasizes reactive behaviors can only propagate new knowledge locally to neighboring modules. He believes that an architecture must have explicit knowledge representation and functionality to allow behaviors to become adaptive and creative (Albus 1991).

2.1.2 Reid Simmons's Architecture

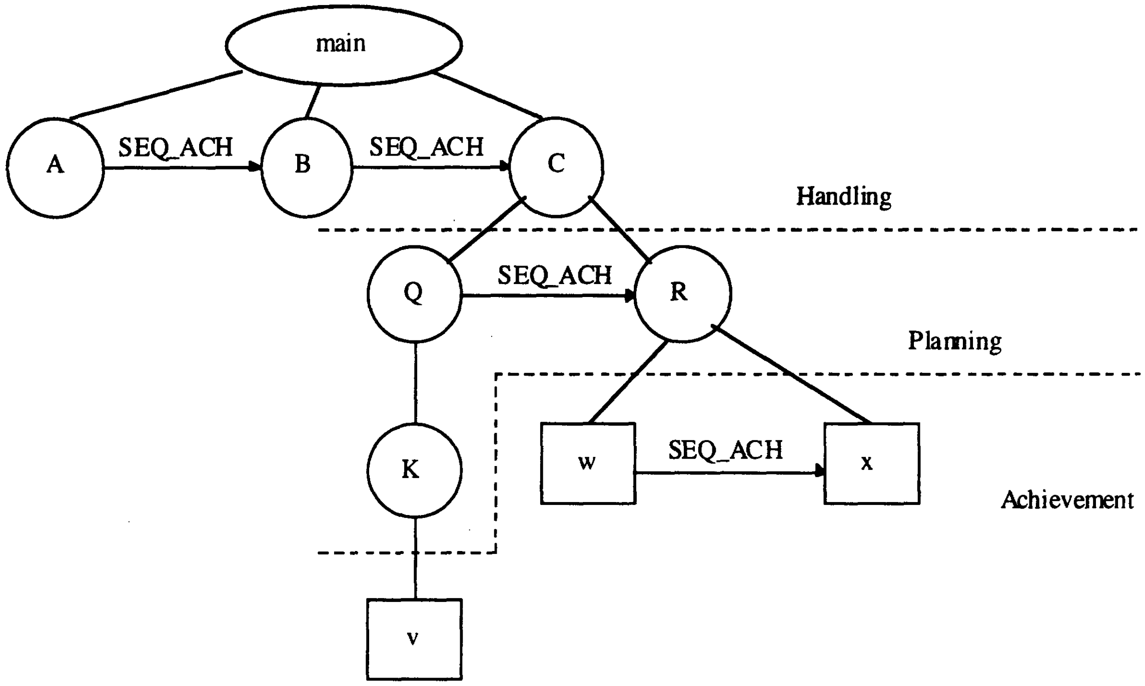

Figure 4 Reid Simmons's Architecture (Simmons et al. 1995).

Reid Simmons proposes an architecture for autonomous robots that emphasizes performing deliberative behaviors, but contains mechanisms for performing reactive behaviors. His architecture, shown in figure 4, is called the Task Control Architecture (TCA). TCA performs deliberative behaviors by representing high level commands, like commands A, B, and C in figure 4, as task trees. Each TCA task tree is decomposed by that task's functionality. For example, the root node of the task tree corresponds to the high level command. The children of the root node correspond to the sub-tasks of the high level command. And the task trees leaves correspond to the low level commands for achieving the task. Different constraints, such as SEQUENTIAL ACHIEVEMENT, can be placed between task tree nodes to achieve desired behavior between task handling, planning, and achievement. TCA contains two mechanisms, monitors and exception handlers, for performing reactive behaviors. During task tree execution, TCA can use monitors to detect unexpected events, and can use exception handlers to deal with faults. TCA uses task-oriented approach to control robots. (Simmons 1994).

Reid Simmons's reason for designing an architecture that emphasizes performing deliberative behaviors is because the "control of planning, perceptions, and action must be well-structured for general-purpose robots to succeed in rich and uncertain environment"

(1994). TCA provides "mechanisms that can map directly from design decisions to methods of communication, task coordination, and reactivity" (Simmons 1994).

2.1.3 James Firby's Architecture

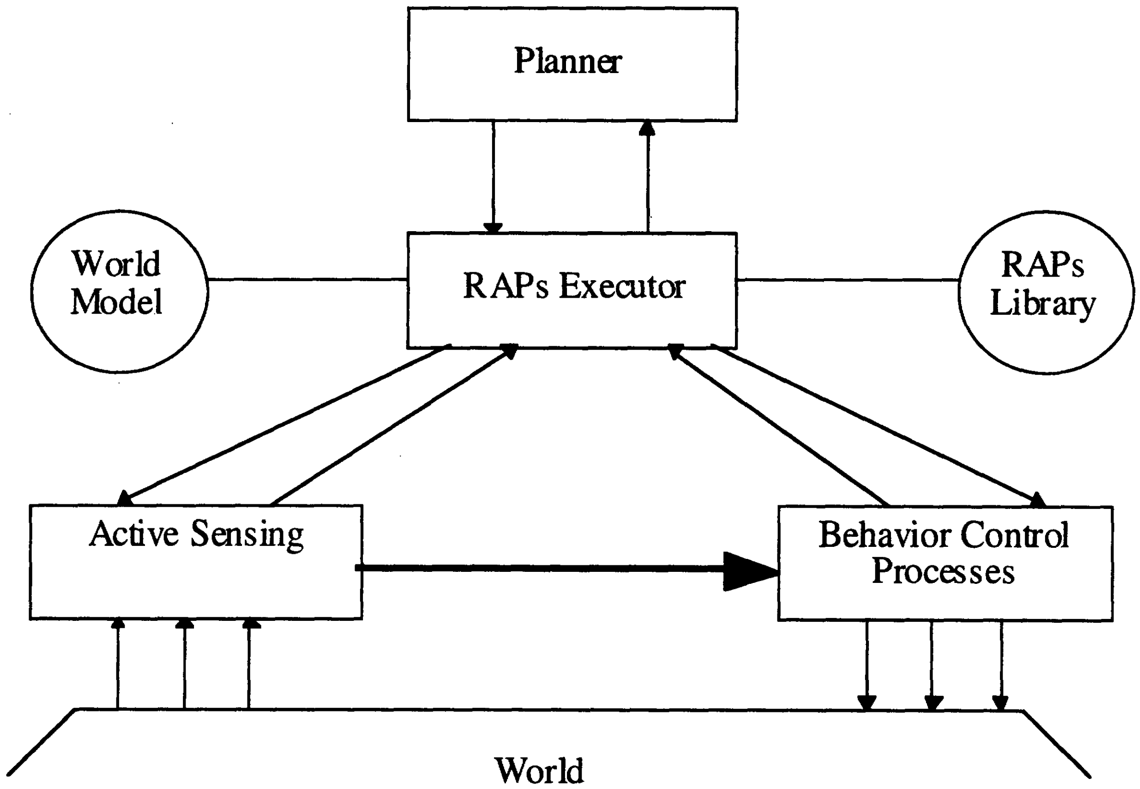

Figure 5 James Firby's Architecture (1994).

James Firby proposes an architecture for autonomous robots that emphasizes performing reactive behaviors, but contains mechanisms for performing deliberative behaviors. The architecture, shown in figure 5, uses three layers to perform deliberative behaviors. The top layer, the Planner, provides sketchy plans for carrying out deliberative behaviors. The middle layer, the Reactive Action Packages (RAPs) Executor, executes sketchy plans from the Planner by using the Active-Sensing and Behavior-Control Processes. The bottom layer, the Active Sensing and Behavior-Control Processes, interfaces with the actual sensors and actuators. The key layer is the RAPs Executor, it performs reactive behaviors by muddling through sketchy plans, and by trying different methods to achieve a task. A sketchy plan consists of several tasks. Each task has several methods for achieving its objective. All unsatisfied tasks are placed in a queue. The RAPs Executor continuously executes the first task in the queue using one of that task's methods. After each execution, the RAPs Executor re-prioritizes tasks in the queue based on the current situation. The RAPs Executor can react to unexpected events by inserting high priority tasks into the queue so that they will be executed next. The RAPs Executor allows fast reaction without deliberation to achieve highly reactive behaviors (Firby 1989).

James Firby's reason for designing an architecture that emphasizes performing reactive behaviors is because the Planner can only provide sketchy plans during planning. Details of the plans can only be filled in during execution time by reacting to the current situation (Firby 1989).

2.1.4 Rodney Brooks's Architecture

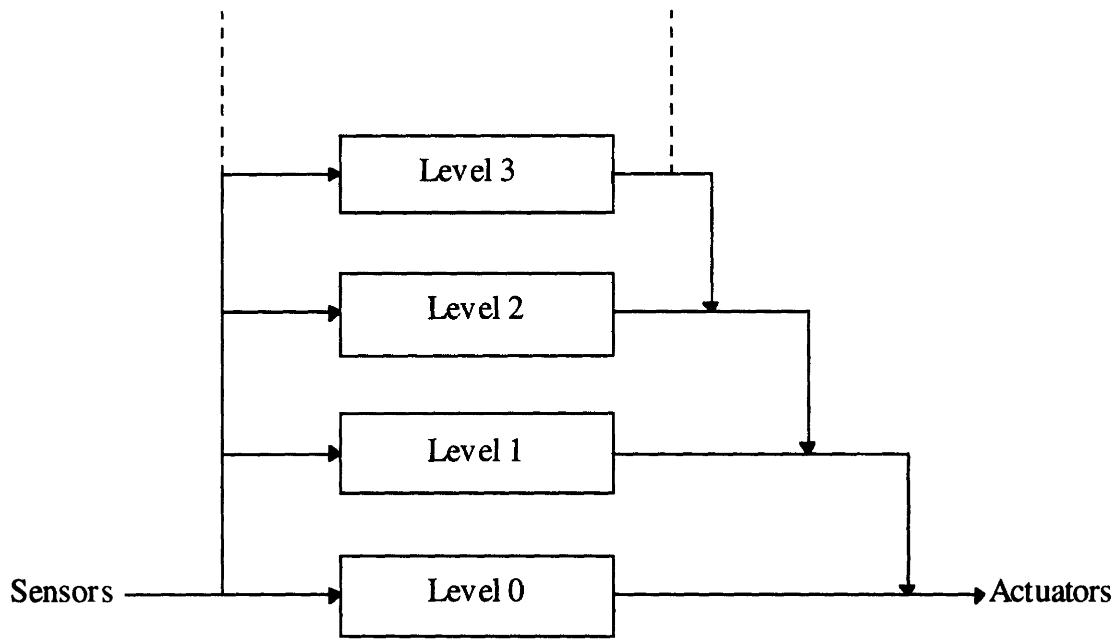

Figure 6 Rodney Brook's Architecture (1985).

Rodney Brooks proposes an architecture for autonomous robots that emphasizes mainly on performing reactive behaviors. His architecture, shown in figure 6, is called the Subsumption architecture. The Subsumption architecture decomposes the control problem by behaviors rather than by functional units, and organizes behaviors into several levels. The architecture has no centralized control, each behavior reacts on its own to the environment. But higher level behaviors can subsume lower level behaviors by blocking lower level behaviors' outputs. The architecture uses bottom-up approach to build lower level behaviors first before building higher level behaviors (Brooks 1985).

Rodney Brooks's reason for designing an architecture that emphasizes reactive behaviors is because of robustness. Lower level behaviors of the Subsumption architecture can still function even if higher level behaviors fail. Organizing control by behaviors greatly increases a robot's robustness. The Subsumption architecture also "stresses reactivity, concurrency, and real-time control" (Ferrell 1993).

2.2 Autonomous Spacecraft Architectures

The Jet Propulsion Laboratory (JPL) has been working on several projects to build autonomous spacecraft. Two recent JPL projects, the Autonomous Serendipitous Science Acquisition for Planets (ASSAP) and the Deep Space 1 (DS1) projects, have made significant contribution in autonomous spacecraft architectures.

2.2.1 ASSAP Architecture

The ASSAP project goal is to demonstrate and integrate new spacecraft technologies for planet mapping missions. The ASSAP project uses the science module to detect surface features, and commands the navigation module to take more pictures of the features. During the mission, the project demonstrates onboard navigation maneuvers such as momentum dumping and drag-makeup maneuvers. The project also proposes several fault protection strategies for autonomous spacecraft. The ASSAP project demonstrates key autonomous spacecraft concepts like "autonomous task planning, sequencing, execution, and recovery from failures" (Aljabri et al. 1996).

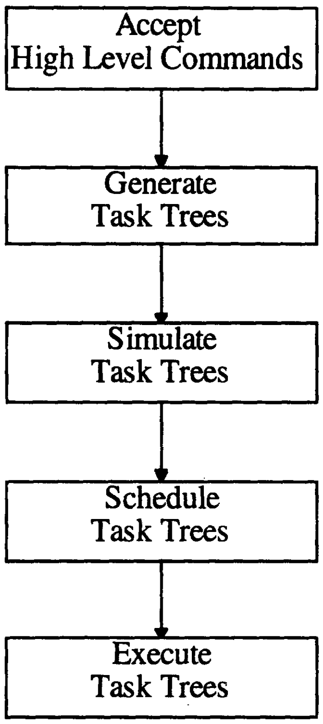

Figure 7 ASSAP Architecture.

The ASSAP architecture uses Reid Simmons's TCA to control spacecraft. The architecture consists of five stages, shown in figure 7 (Shih 1995). First, it accepts high level commands from a Planner. Second, it generates TCA task trees to accomplish these commands. Third, it simulates these TCA task trees to validate them. Fourth, it schedules TCA task trees based on their resource requirements. Finally, it uses TCA task management to execute task trees. The ASSAP project demonstrates that TCA is very useful for controlling autonomous spacecraft.

2.2.2 DS1 Architecture

The state-of-the-art software architecture for autonomous spacecraft is the DS1 architecture. The DS1 spacecraft is scheduled to be launched in 1998 to validate onboard autonomous control of spacecraft. The architecture, designed by Pell et al., can support six activities that are usually done on the ground. They are "planning activities, sequencing spacecraft actions, tracking spacecraft state, ensuring correct functioning, recovering in cases of failures, and reconfiguring hardware". Their architecture "integrates traditional real-time monitoring and control with constraint-based planning and scheduling, robust multi-threaded execution, and model-based diagnosis and reconfiguration" (Pell et al. 1996).

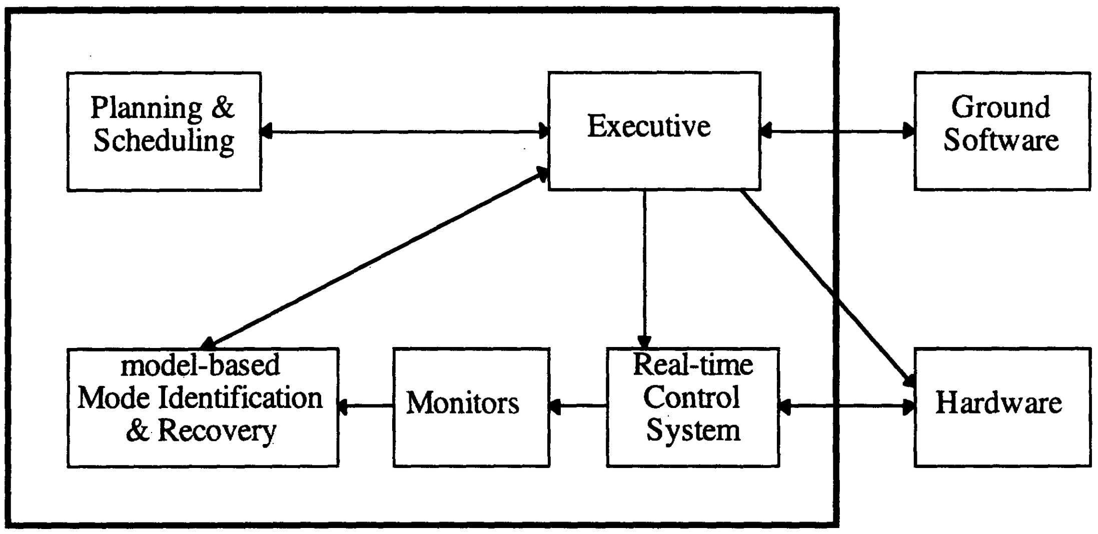

Figure 8 DS1 Architecture (Pell et al. 1996).

The DS1 architecture, shown in figure 8, consists of five modules. They are the Planning-and-Scheduling, Executive, Real-time-Control-System, Monitors, and model-based-Mode-Identification-and-Recovery (MIR) modules. The Planning-and-Scheduling module provides high level plans to the Executive module. The Executive module carries out high level plans by using task trees to connect domain specific modules such as the navigation and guidance-control modules. The Executive module then uses Real-time-Control-System module to execute task trees on the Hardware. The Monitors and MIR modules are used to support reactive behaviors. The Monitors module detects faults and the MIR module produces fault recovery procedures. The DS1 architecture uses many ideas in Reid Simmons's and James Firby's architectures to control autonomous spacecraft.

Pell et al. mention that the main drawback of their architecture is lack of a consistent knowledge database. The Planning-and-Scheduling, Executive, and MIR modules all have their own database where they store information about the world. Therefore, the architecture can have three different views of the world (Pell et al. 1996).

3 ASPIRE Architecture Framework

3.1 Spacecraft Requirements

Pell et al. mention six design requirements for autonomous spacecraft. First, autonomous spacecraft need to meet hard deadlines. Missing the time frame for maneuvers, such as orbit insertion, may jeopardize spacecraft health or may waste unnecessary fuel. Therefore, spacecraft need to define a global timing concept to meet hard deadlines. Second, autonomous spacecraft have tight resource constraints. All science instruments may have to be turned off during engine firing to conserve power usage. Hence, spacecraft need to share and use resources efficiently. Third, autonomous spacecraft have limited observability. Each sensor adds weight, so only sensors that have clear values are placed on the spacecraft. As the result, spacecraft should maximize the use of all available sensor information when making decisions. Fourth, autonomous spacecraft need to perform concurrent activities. Multiple events may require spacecraft's attention at the same time. Therefore, spacecraft need to handle them concurrently. Fifth, autonomous spacecraft need to support long operation periods. With many spacecraft exploring the solar system, the Deep Space Network antennas can only communicate with each spacecraft for a small time period. Hence, spacecraft need to degrade gracefully when faults occur between ground communications. Sixth, autonomous spacecraft need to have high reliability. All spacecraft components must be extremely reliable. As the result, spacecraft need to use additional software and hardware to achieve high reliability. These six spacecraft requirements are important for designing autonomous spacecraft architectures (1996).

3.2 Design Philosophy

Reid Simmons's task-orientated architecture that emphasizes performing deliberative behaviors with added mechanisms for performing reactive behaviors is the most appropriate architecture, of the four autonomous robot architectures in chapter 2, for controlling autonomous spacecraft. An architecture that emphasizes performing deliberative behaviors can handle hard deadlines, tight resource constraints, limited observability, and concurrent activities requirements. Added mechanisms for performing reactive behaviors can support long operation periods and high reliability requirements. Therefore, Reid Simmons's architecture can support all six spacecraft requirements in section 3.1.

A deliberative architecture is more appropriate for controlling autonomous spacecraft than a reactive architecture. Most of the time, spacecraft are placed in predictable environments conducting predictable behaviors. Rarely do they need fast response time to handle unexpected events. Reactive architectures, like James Firby's RAPs and Rodney Brook's Subsumption architecture, achieve fast response time by assuming execution without context has no bad consequences. But, muddling through sketchy plans in RAPs or using highly reactive behaviors in the Subsumption architecture can cause spacecraft to miss hard deadlines or waste resources. Hartley and Pipitone also point out further problems with the Subsumption architecture that make it hard to control complexity (1991). Therefore a deliberative architecture is more appropriate for controlling autonomous spacecraft than a reactive architecture.

A task-oriented deliberative architecture, like Reid Simmons's TCA, is more appropriate for controlling autonomous spacecraft than a time-oriented deliberative architecture like James Albus's architecture (Simmons 1994). Organizing an architecture by task's functionality, like TCA, allows easier reasoning and implementation. Therefore, the most appropriate architecture philosophy for autonomous spacecraft is a task-oriented architecture that emphasizes performing deliberative behaviors with added mechanisms for performing reactive behaviors.

3.3 Architecture Framework

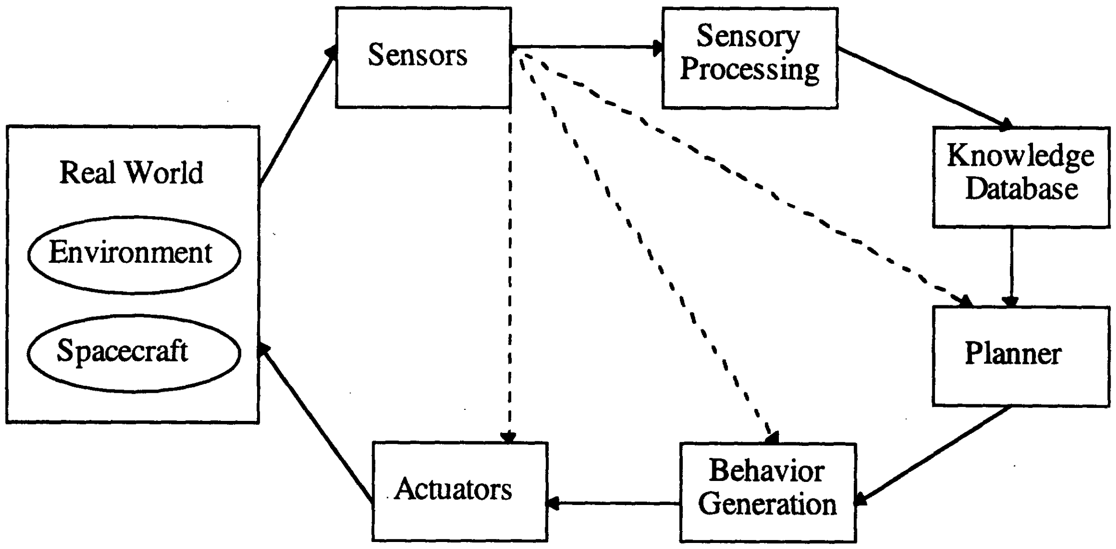

Figure 9 ASPIRE Architecture Framework.

The ASPIRE architecture framework, shown in figure 9, consists of a deliberative path and three bypass paths. The deliberative path is used for conducting deliberative behaviors, while the three bypass paths are added for conducting reactive behaviors. The deliberative path is similar to James Albus's hierarchy node architecture. But with the three bypass paths, the architecture framework looks similar to Rodney Brook's Subsumption architecture.

3.3.1 Deliberative Path

The deliberative path consists of the six modules shown in figure 9. They are the Sensors, Sensory-Processing, Knowledge-Database, Planner, Behavior-Generation, and Actuators modules. First, the Sensors module provides data about the Real-World to the Sensory-Processing module. The Sensory-Processing module then interprets data and stores them in the Knowledge-Database module. All the modules can access information in the Knowledge-Database module. Next, the Planner module generates high level commands and stores them in an execution queue. When appropriate, high level commands in the execution queue are sent to the Behavior-Generation module. The Behavior-Generation module then uses pre-planned task trees to generate task trees for carrying out high level commands. Finally, the Behavior-Generation module executes task trees' low level commands on the Real-World using the Actuators module. The deliberative path reacts to unexpected events by having the Planner module re-prioritizes the tasks in the execution queue. In summary, the Planner module uses the deliberative path to carry out deliberative behaviors.

3.3.2 Three Bypass Paths

The three bypass paths, shown in figure 9, are added for performing reactive behaviors. The first bypass path connects the Sensors module directly with the Actuators module. The second bypass path connects the Sensors module with the Behavior-Generation module. The third bypass path connects the Sensors module with the Planner module. With the three bypass paths, the architecture framework can be viewed as a Subsumption architecture. The first bypass path can be viewed as the first level of the Subsumption Architecture. The second bypass path can be viewed as the second level. The third bypass path can be viewed as the third level. And the deliberative path can be viewed as the fourth level. Each higher level path provides more deliberation, but slower response time between the Sensors module and the Actuators module. Higher level paths of the architecture can subsume lower level paths when they have time to handle events.

The first bypass path connects the data from the Sensors module directly to the Actuators module. It allows the Actuators module to immediately react to unexpected events. This path's goal is to ensure survival by providing fast reaction without deliberation. Thus, the first bypass path skips all the software modules to control the spacecraft.

The second bypass path connects the data from the Sensors module to the Behavior-Generation module. It allows the Behavior-Generation module to interrupt the current task execution. Interrupts can only occur between the low level commands send to the Actuators module. When an interrupt occurs, all planning for the current task execution are discarded. A cleanup task tree is first executed to halt the current task execution. Then an emergency task tree is executed to deal with the interrupt. Afterwards, a re-planning task tree is executed to resume the interrupted task. The second bypass path skips the Planner module to control the spacecraft.

The third bypass path connects the data from the Sensors module to the Planner module. It allows the Planner module to abort the current command and issue a new command. Aborting the current command takes more time than interrupting the current task execution because of the need to perform a complete cleanup. The third bypass path skips the Sensory-Processing module to control the spacecraft.

3.3.3 Illustrations

The deliberative path can support all types of deliberative behaviors for autonomous spacecraft. The Sensory-Processing module can detect surface features, surface changes, and ejected fragments. The Sensory-Processing module can also estimate spacecraft position, spacecraft state, and internal comet model. Similarly, the Behavior-Generation module can execute momentum dumping, drag-makeup, orbiting, and close flyby maneuvers. The Behavior-Generation module can also track ejected fragments and surface targets. The deliberative path can perform all the deliberative behaviors required by the ASPIRE project.

The three bypass paths can provide faster response time for the spacecraft. For example, when a cometary fragment is coming toward the spacecraft, the deliberative path can plan a maneuver to move away. But if faster response time is needed, the third bypass path can abort the current command to execute an escape maneuver. If faster response time is needed, the second bypass path can interrupt the current task execution to perform an escape maneuver. If a very fast response is needed, the first bypass path can control the thrusters to move the spacecraft away. Another example is when the thrusters are not working correctly during a burn. The deliberative path can adapt high level commands to avoid the faulty thrusters. If faster response time is needed, the third bypass path can abort the current burn, perform the thruster shut-off sequence, and analyze the problem immediately. If faster response time is needed, the second bypass path can interrupt the burn, turn off the faulty thrusters, reconfigure the backup thrusters, recalculate the needed maneuver, and resume the burn. If a very fast response is needed, the first bypass path can shut off the faulty thrusters immediately. The three bypass paths can provide faster response time for dealing with unexpected events.

3.4 Evaluation

This architecture framework for autonomous spacecraft contains seven good design elements. First, it integrates deliberative behaviors with reactive behaviors by using the deliberative path to perform a task when there is enough time, and by using the three bypass paths when faster response time is needed. Second, it provides one consistent view of the Real-World to all the software modules by using one common database for storing all the information. Third, it provides fault protection by using the deliberative path and the three bypass paths to increase the robustness between the Sensor module and the Actuator module. Fourth, it allows different pre-planned plans to be combined by using stateless task trees to represent them and by using the Knowledge-Database module to store all the state information during task tree execution. Fifth, it separates high level planning from real-time control by using the Planner module to issue high level commands and by using the Behavior-Generation module to execute them. Sixth, it demonstrates the use of additional software for improving existing capabilities by using the three bypass paths to improve the response time of the deliberative path. Seventh, it provides the spacecraft software an easy interface with the spacecraft hardware by using TCA task trees to represent and execute high level commands on the hardware. These seven design elements are very useful for designing autonomous spacecraft architecture.

3.5 Issues

There are two issues with the architecture framework. The first issue is whether there should be other bypass paths. For example, a bypass path connecting the Sensory-Processing module to the Behavior-Generation module will give the Sensor-Processing module direct control of the Behavior-Generation module. The second issue is whether there should be feedback paths. For example, a feedback path from the Behavior-Generation module back to the Planner module will provide the Planner module direct feedback from the Behavior-Generation module. Adding more bypass and feedback paths will make the architecture look more reactive.

3.6 Future Work

The architecture framework needs more work in five areas. First, the framework needs to define how high level paths can subsume lower level paths. The higher level paths need to take control when they can react to events, and need to relinquish control when they cannot. Second, the framework needs to define how to handle prioritization in each bypass path. Prioritization in each bypass path is needed for dealing with multiple aborts and nested interrupts. Third, the framework needs to make the Knowledge-Database module very reliable. The Knowledge-Database module must be reliable because all the software modules use it. Fourth, the framework needs a sensory processing module to support fault detection. The Cassini spacecraft uses a rule-based system for fault detection and the DS1 spacecraft uses a model-based system for fault detection (Pell et al. 1996). Fifth, the framework needs a behavior generation module to support fault recovery. Mechanisms are needed for handling software and hardware faults. Further work in these five areas will make the architecture framework more complete.

4 ASPIRE Implementation

4.1 Overview

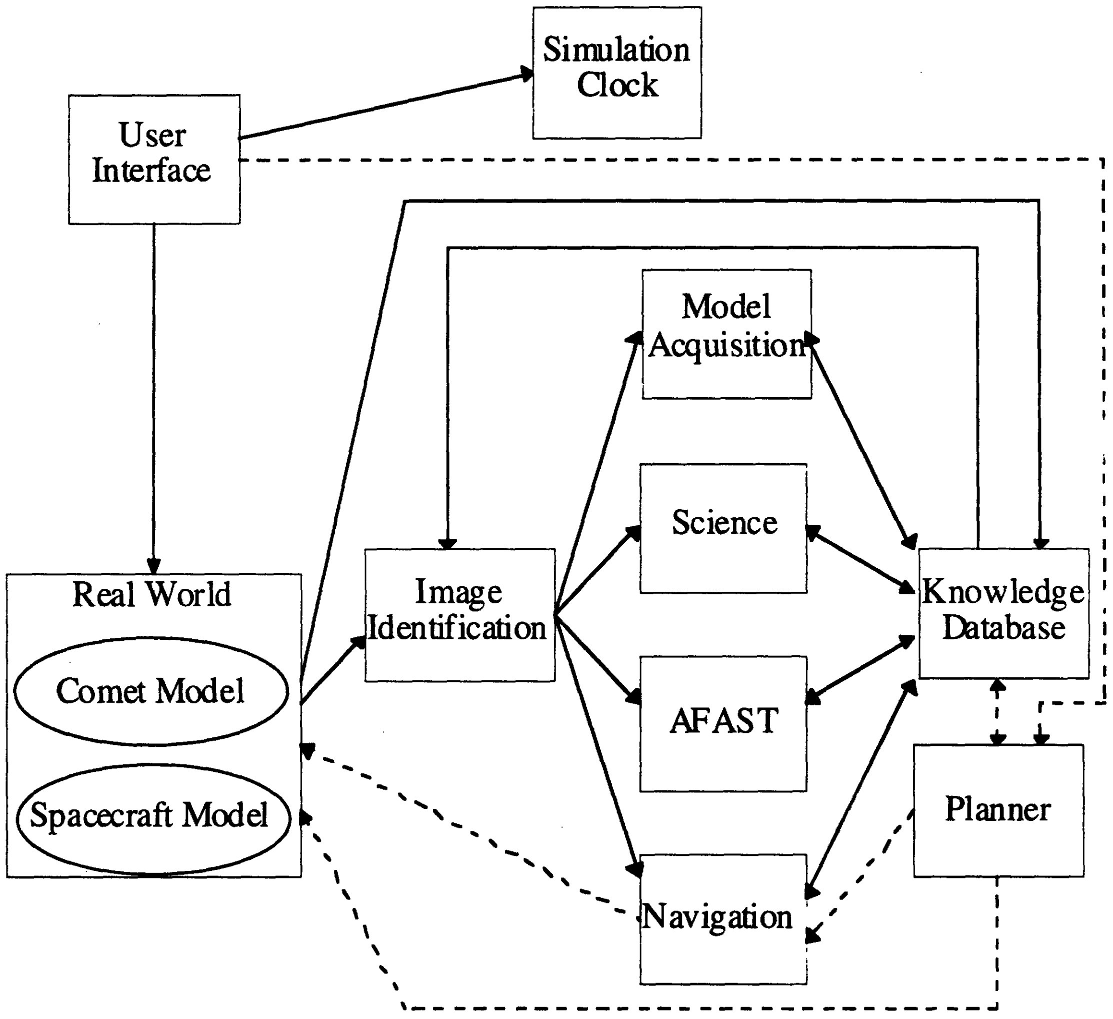

Figure 10 ASPIRE Implementation.

The ASPIRE implementation is based on the architecture framework discussed in chapter 3 of using the deliberative path for performing deliberative behaviors and using the three bypass paths for performing reactive behaviors. The deliberative path, shown in figure 10, consists of ten modules. They are the Real-World, Image-Identification, Model-Acquisition, Science, AFAST, Navigation, Knowledge-Database, Planner, Simulation-Clock, and User-Interface modules. TCA messages are used for communication between modules. TCA task trees are used to represent and execute

Planner's high level commands. The three bypass paths are not implemented. Nevertheless, the deliberative path can support all the deliberative behaviors required by the ASPIRE project.

The deliberative path consists of two parts, sensing and acting. The goal of the sensing part is to gather information about the Real-World module's comet and spacecraft models. The Real-World module outputs camera images to the Image-Identification module for processing. The Image-Identification module then determines locations on the comet for these images using the Knowledge-Database module's internal comet model. Afterwards, the Image-Identification module outputs identified images to the Model-Acquisition, Science, AFAST, and Navigation modules. The Model-Acquisition module uses these images to update the Knowledge-Database module's internal comet model. The Science module uses these images to detect sub-pixel level movement. The AFAST module uses these images to detect and track pixel level movement. Finally, the Navigation module uses these images to estimate spacecraft position. All information from these four sensory processing modules are stored in the Knowledge-Database module. The Knowledge-Database module also receives information about the camera position from the Real-World module's spacecraft model. Information stored in the Knowledge-Database module can be accessed by all the modules. The sensing part of the deliberative path uses many sensing algorithms to estimate the environment.

The goal of the acting part is to control the spacecraft for capturing scientific events. The Planner module can query the Knowledge-Database module for a list of science targets. The Planner module can command the Real-World module to track these targets using the narrow-field camera. The Planner module can also command the Navigation module to produce plans for performing close flyby maneuvers over them. Commands and plans from the Planner and Navigation modules are executed on the Real-World module's spacecraft model. The acting part of the deliberative path uses many control algorithms to control the spacecraft.

The Simulation-Clock and User-Interface modules are used to support the software simulation. The Simulation-Clock module simulates the clock onboard the spacecraft by broadcasting the current time to all the modules. The User-Interface module can perform three functions. It can simulate ground commands to the Planner module, can inject changes to the Real-World module's comet model, and can change the broadcast interval in the Simulation-Clock module. In the actual spacecraft, the Simulation-Clock module will be replaced by a real clock and the User-Interface module will be replaced by a module accepting ground commands.

4.2 Module Description

The following is a brief description of each software module.

4.2.1 Real-World Module

The Real-World module contains two coordinate frames for simulating the environment. The first one is the INERTIAL frame and the second one is the COMET-FIXED frame. These two frames share the same origin. At time 0, the x axis of the COMET-FIXED frame is in the x-y plane of the INERTIAL frame, and rotates toward the positive z axis of the INERTIAL frame. The Real-World module simulates the environment using a comet model and a spacecraft model. The comet model is four kilometers wide in diameter with a constant density of 1000.0 kilograms per meter cube. The comet model rotates along the z axis of the INERTIAL frame with a constant rotational rate of one revolution per two days. The User-Interface module can inject various changes to the comet model. The spacecraft model, on the other hand, contains two cameras and several thrusters. The first camera is a 2 degree by 2 degree narrow-field camera and the second camera is a 20 degree by 20 degree wide-field camera. Both cameras take 512 by 512 pixel images. The narrow-field camera can be controlled by the Planner module to take fine resolution pictures. The wide-field camera normally points at the comet center and takes a picture every 20.0 seconds. The Navigation module can control the wide-field camera to take images for landmark measurements. The thrusters consist of one main engine for performing delta-V maneuvers and several small thrusters for performing close flyby maneuvers. These thrusters are all controlled by the Navigation module. In summary, the architecture uses the Real-World module to simulate the environment.

4.2.2 Image-Identification Module

The Image-Identification module determines the coordinates on the comet for each camera image. Its function is to inform other modules where on the comet each image is looking at. It queries the Knowledge-Database module for the internal comet model and the spacecraft position to help it identify images using comet landmarks. The Image-Identification module sends identified images to the Model-Acquisition, Science, AFAST, and Navigation modules for further processing.

4.2.3 Model-Acquisition Module

The Model-Acquisition module receives images from the Image-Identification module and updates the Knowledge-Database module's internal comet model. The Model-

Acquisition module performs four functions. The first function is determining comet's rotational rate and rotational axis. The second function is constructing comet's shape with a wire-frame model. The third function is constructing a texture map of the comet. Finally, the fourth function is storing locations of comet landmarks. Essentially, the Model-Acquisition module estimates the Real-World module's comet model.

4.2.4 Science Module

The Science module receives wide-field camera images from the Image-Identification module. It uses a change detection algorithm to compare them with past images of the same location to detect sub-pixel level movements. When the Science module detects a sub-pixel level movement, it informs the Knowledge-Database module about the center and size of the movement. The Science module can detect sub-pixel level movements, such as cracks, occurring on the comet.

4.2.5 AFAST Module

The AFAST module receives wide-field camera images from the Image-Identification module. It compares each image with the previous image to detect pixel-level movements. If the AFAST module is not tracking a movement, it filters all the detected pixel level movements through a detection threshold. If there are movements greater than the threshold, it informs the Knowledge-Database module about the center and size of the largest movement, and starts tracking it. The AFAST module can detect pixel level movements such as ejected particles coming out from the comet.

4.2.6 Navigation Module

The Navigation module performs both sensory processing and behavior generation functions. For sensory processing, it can control the wide-field camera to take images for landmark measurements. After receiving these images from the Image-Identification module, the Navigation module generates a spacecraft position profile estimating the spacecraft position. For behavior generation, the Navigation module produces plans for performing different types of maneuvers, such as orbiting, close-flyby, and escape to safety maneuvers, around the comet. It also informs the Knowledge-Database about the status of the current maneuver. The Navigation module can query the Knowledge-Database module for information, such as comet's rotational rate, rotational axis, and wire-frame model, to help it navigate. The Navigation module is responsible for estimating and controlling spacecraft position.

4.2.7 Knowledge-Database Module

The Knowledge-Database module stores and updates information about the Real-World module. The Model-Acquisition module updates comet's rotational rate, rotational axis, wire-frame model, texture map, and landmark locations. The Science and AFAST modules report locations of interesting targets on the comet. The Navigation module updates spacecraft position profile and maneuver status. The Real-World module updates current camera position. When the Science and AFAST modules discover a target, the Knowledge-Database module converts the target location on the image to the target location in the internal comet model. Afterwards, the Knowledge-Database module will inform the Planner module that a new target has been discovered. The Knowledge-Database provides one consistent view of the Real-World module for all the software modules.

4.2.8 Planner Module

The Planner module controls the spacecraft by generating high level commands. It can generate commands to control the narrow-field camera, or to perform different maneuvers around the comet. The Planner module can also generate commands in response to the ground commands from the User-Interface module. All commands are placed in an execution queue, and are sent to the Real-World and Navigation modules when appropriate time comes. The Planner module decides what actions the spacecraft should take in response to the information in the Knowledge-Database module.

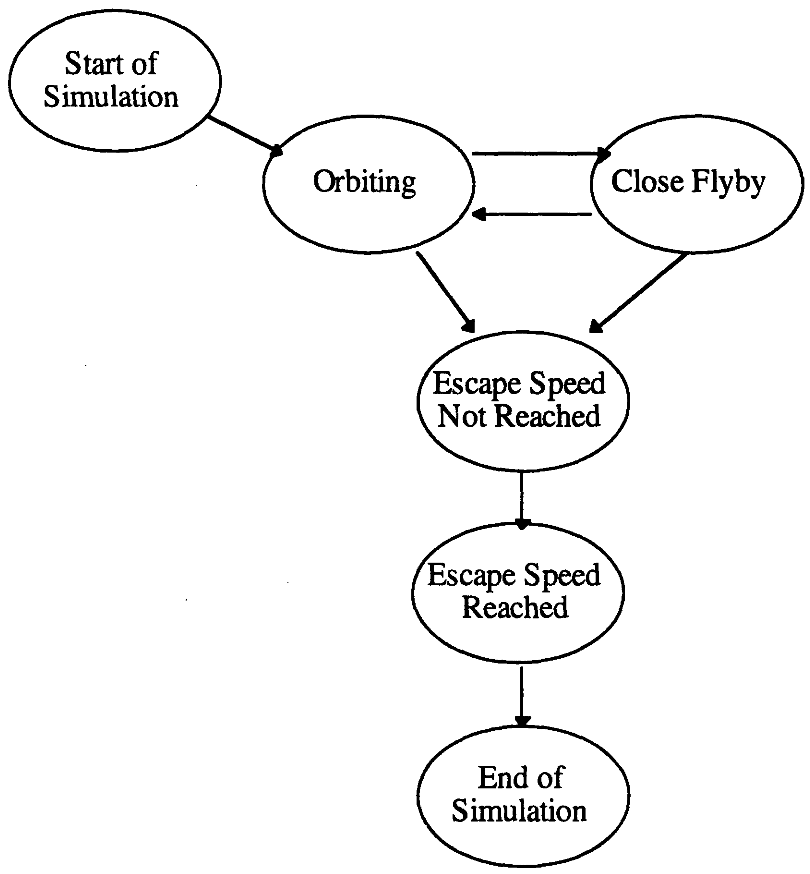

Figure 11 State Transition Diagram.

The Planner module uses the state transition diagram, shown in figure 11, to control the spacecraft movement. The spacecraft is initially in the Orbiting state. The Planner module can find out the current state from the Knowledge-Database module, and can use different commands to change it. If the spacecraft is in the Orbiting state, the Planner module can issue a close flyby command to enter the Close-Flyby state. After a close flyby maneuver is performed, the spacecraft returns to thc Orbiting state. When one of the sensory processing module detects an emergency, such as a comet breakup or ejected particles moving toward the spacecraft, the Planner module can issue an escape to safety command to slowly move the spacecraft away from the comet. The simulation ends when the spacecraft reaches the escape speed. The Planner module can capture scientific events by issuing different high level commands to the spacecraft.

4.2.9 Simulation-Clock Module

The Simulation-Clock module simulates the clock onboard the spacecraft. It broadcasts the current time in seconds to all the modules. It can also receive queries from other modules for the current time. The Simulation-Clock module has two mode. During the SLOW mode, it broadcasts the time every 20.0 seconds, and during the FAST mode, it broadcasts the time every 30.0 minutes. The Simulation-Clock module performs discrete-time simulation by broadcasting the next time only after all the modules have finished with their processing for the current time. The User-Interface module can change the broadcast interval to speed up or slow down the simulation. The Simulation-Clock module represents the real clock onboard the spacecraft.

4.2.10 User-Interface Module

The User-Interface module's primary function is to allow the user to inject events and commands. For events, the user can inject cracks on the comet, can eject particles from the comet, or can break the comet apart. For commands, the user can request the Planner module to perform a close flyby, an orbiting, or an escape to safety maneuver. Finally, the User-Interface module can change the Simulation-Clock module's broadcast interval to speed up or slow down the simulation. The User-Interface module is used mainly for supporting the simulation.

4.3 Evaluation

The ASPIRE implementation has ten good elements. First, two coordinate frames are used to specify positions. All spacecraft, camera, and target positions are described in both the INERTIAL frame and the COMET-FIXED frame. Some modules prefer positions described in the INERTIAL frame while other modules prefer positions described in the COMET-FIXED frame. Having all position vectors described in both frames provides an easy solution for module communication.

Second, camera images are classified by how they are taken instead of when they are taken. Each image is tagged with a time stamp, a camera type, a spacecraft trajectory vector, a camera bore-sight vector, a camera orientation vector, and a sun position vector instead of a mapping number, an orbit number, and a picture number. The former tags describes the exact viewing location on the comet for each image.

Third, science targets are identified by vectors in the internal comet model. The Science and AFAST modules detect movements on the images. Since images are classified by how they are taken, the Knowledge-Database module can convert the targets on the images to the target vectors in the internal comet model. Representing science targets by vectors in the internal comet model provides a simple interface for the Planner module to respond to new target discoveries.

Fourth, each interesting target contains an image patch of the target. By using the narrow-field camera, the Real-World module can use these image patches to better track targets. Image patches provide useful information about the discovered targets.

Fifth, the Planner modules uses a state transition diagram to control the spacecraft. The Planner module uses high level commands to control the spacecraft by changing the spacecraft state. State transition diagram allows easy implementation and modification of the Planner module.

Sixth, the camera resource is shared between several modules. The Model-Acquisition, Science, AFAST, and Navigation modules share one wide-field camera and one narrow-field camera. Two cameras are sufficient to meet the needs of these four modules.

Seventh, all software modules use one internal comet model. An internal comet model that contains comet's rotational rate, rotational axis, wire-frame model, texture map, and landmark locations can meet the all the software modules' needs. Having one internal comet model provides one consistent view of the Real-World module.

Eighth, message passing is used for module communication. Message passing is better than procedure call for module communication because it forces the programmer to define a simple and clear interface. Message passing provides good abstraction for each module.

Ninth, blocking messages are used for all the message communication. Blocking messages are better than non-blocking messages for module communication because they force the programmer to return from message handlers immediately. It is dangerous to mix blocking and non-blocking messages in an architecture. Using all blocking messages for module communication provides a good way to reason the architecture.

Tenth, the number of messages and the data associated with each message should be kept to the minimum. A large number of messages or large message datum complicates the interface. Each message should provide a clear function. Fewer messages and smaller message data provide better interface.

4.4 Testing

The ASPIRE implementation tests all the autonomy software by injecting various changes to the comet model during the mission scenario. The Science module is tested by injecting different sub-pixel level movements at different locations on the comet model. Similarly, the AFAST module is tested by ejecting different particles from different locations on the comet model. When science targets are detected, the Planner module autonomously controls the narrow-field camera to take more pictures of them, and autonomously commands the Navigation module to perform close flyby maneuvers over them. The ASPIRE project demonstrates successful integration of these technology modules.

4.5 Issues

There are three issues with this architecture implementation. First, should the Simulation-Clock module use discrete-time or real-time simulation. Discrete-time simulation eliminates many important timing issues but does allow time scale to change. Second, how should the architecture deal with computation power and communication bandwidth limits. Modules performing urgent tasks need higher priority when using these scarce resources. Third, how can we make distributed programming easier. Mechanisms are needed for dealing with timing issues associated with message passing. These three issues are important for the architecture implementation.

4.6 Future Work

The architecture implementation can improve in two more areas, beside the five areas already mentioned in section 3.6. First, it can simulate task tree execution in the Planner module to increase the confidence of the plans. Second, it can use a more sophisticated spacecraft model for simulation. These two areas and the five areas mentioned in section 3.6 are needed to make the implementation complete.

5 Conclusion

Autonomous spacecraft can use deliberative behaviors to achieve missions objectives and reactive behaviors to handle unexpected events. The ASPIRE architecture framework uses the deliberative path to perform deliberative behaviors, and the three bypass paths to perform reactive behaviors. The ASPIRE architecture framework looks like a Subsumption architecture with the four paths connecting the Sensor module with the Actuator module. The deliberative path can subsume the three bypass paths when it has time to handle events. The three bypass paths are used to provide faster response time, but less deliberation, between the Sensor module and the Actuator module. The ASPIRE project shows that the deliberative path can support technology integration for a comet orbiter mission.

In conclusion, a good way for controlling autonomous spacecraft is a task-oriented architecture that emphasizes performing deliberative behaviors with added mechanisms for performing reactive behaviors. A good way for designing autonomous spacecraft architecture is to start with a deliberative path for performing deliberative behaviors and then add bypass paths to make the architecture more reactive. Finally, a good way for implementing the deliberative path is to use one Knowledge-Database module to store all the information about the environment. Therefore, the ASPIRE architecture framework is a good framework for controlling autonomous spacecraft.

Appendix A: Module Interface

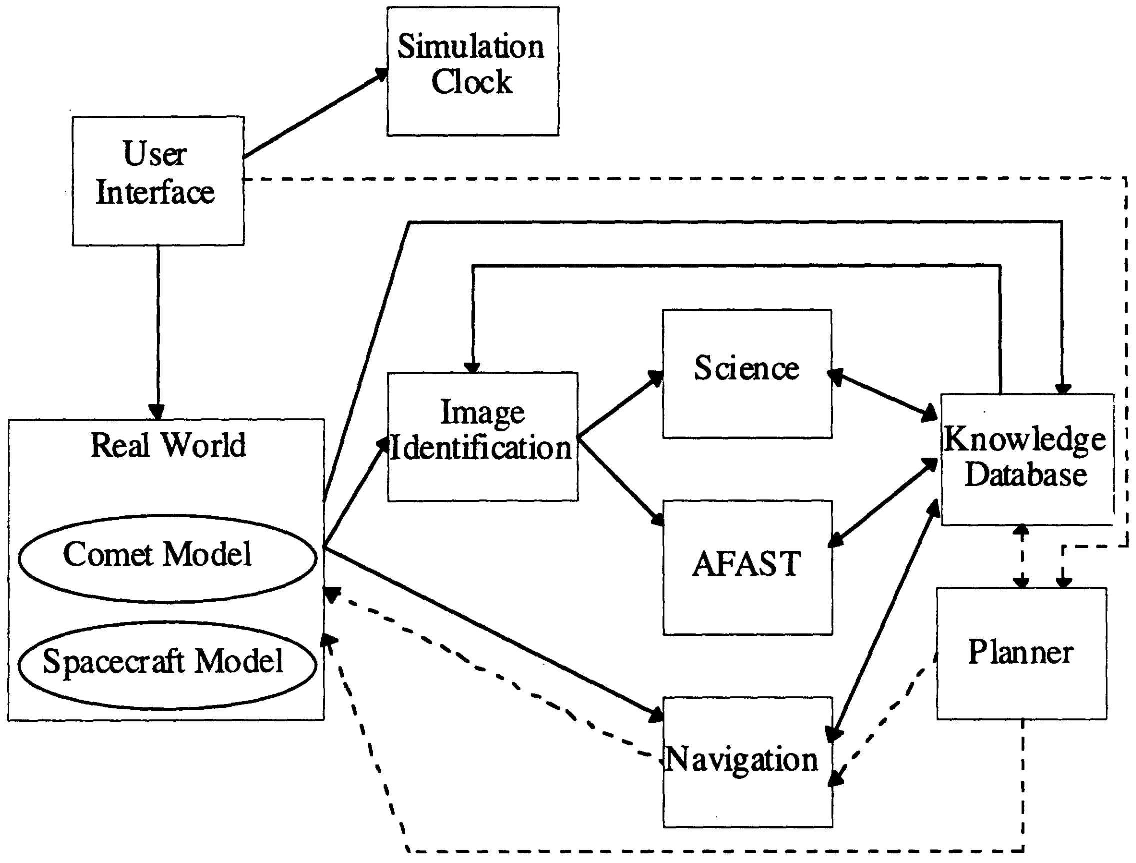

Figure 12 Current ASPIRE Implementation.

The current ASPIRE implementation, shown in figure 12, differs from the complete implementation in three areas. First, the current implementation doesn't have a Model-Acquisition module to determine and update the Knowledge-Database module's internal comet model. Therefore, it assumes that the Knowledge-Database module has a perfect knowledge of the Real-World module's comet model. Second, the Navigation module doesn't perform image processing for determining landmark locations. Instead, the current implementation allows the Navigation module to query the Real-World module for landmark locations in the current camera image. Third, the Image-Identification module doesn't identify images based on landmarks. The current implementation allows the Image-Identification module to use the current spacecraft position to estimate what part of the comet each image is looking at. These three areas are passed over to make the implementation easier.

A.1 Additional Data Types

The following thirteen data types, **sim_time**, **vector**, **deltaV**, **hardware_control**, **landmark**, **spacecraft_position**, **camera_position**,

raw_image, **identified_image**, **point**, **target**, **global_parameters**, and **wire_frame**, are defined to support message communication.

A.1.1 sim_time

typedef **float sim_time**;
The **sim_time** data type contains a non-negative time in seconds. If a **sim_time** variable is equal to -1.0, then it is not defined.

A.1.2 vector

typedef struct {**float** *x*, *y*, *z*;} **vector**;
The **vector** data type contains a vector in kilometers. The *z* axis points up and the *x* axis is on the left side of the *y* axis. If a **vector** variable is equal to (0.0, 0.0, 0.0), then it is not defined.

A.1.3 deltaV

typedef struct {**sim_time** *current_time*; **vector** *deltaV_COMET*;} **deltaV**;
The **deltaV** data type contains a delta-V request. The *current_time* field indicates the time for the delta-V. The *deltaV_COMET* field indicates the delta-V vector in the COMET-FIXED frame.

A.1.4 hardware_control

```
typedef struct {sim_time measurement_requests[10];
                double thrust_constant1;
                double thrust_constant2;
                double thrust_constant3;
                double thrust_constant4;
                double thrust_constant5;
                double thrust_constant6;
                double thrust_constant7;
                double thrust_constant8;
                double thrust_constant9;
                double thrust_constant10;
                double thrust_constant11;
                double thrust_constant12;
                double thrust_constant13;
                double thrust_constant14;
                double thrust_constant15;
                double thrust_constant16;
                double thrust_constant17;
                double thrust_constant18;
                double thrust_constant19;
                deltaV deltaV_requests[10];} hardware_control;
```

The **hardware_control** data type contains a spacecraft hardware control request. The *measurement_requests* field is an array of measurement requests. The *thrust_constant** fields are thruster control constants. The *deltaV_requests* field is an array of delta-V requests.

A.1.5 landmark

typedef struct {**sim_time** *current_time*;

vector *observation_COMET*;
vector *landmark_COMET*;} **landmark**;

The **landmark** data type contains a landmark measurement. The *current_time* field is the measurement time. The *observation_COMET* field is an unit vector from the spacecraft to the landmark in the COMET-FIXED frame. The *landmark_COMET* field is a vector from the comet center to the landmark in the COMET-FIXED frame.

A.1.6 spacecraft_position

typedef struct {**sim_time** *current_time*;
vector *trajectory_INERTIAL*;
vector *thrust_INERTIAL*;
vector *velocity_INERTIAL*;
vector *deltaV_INERTIAL*;
vector *sun_INERTIAL*;
vector *trajectory_COMET*;
vector *thrust_COMET*;
vector *velocity_COMET*;
vector *deltaV_COMET*;
vector *sun_COMET*;} **spacecraft_position**;

The **spacecraft_position** data type contains a spacecraft position in both INERTIAL and COMET-FIXED frames. The *current_time* field is the measurement time. The *trajectory_** field is a vector from comet center to the spacecraft. The *thrust_** field is a vector from the spacecraft indicating the thrust. The *velocity_** field is a vector from the spacecraft indicating the velocity. The *deltaV_** is a vector from the spacecraft indicating the delta-V. The *sun_** field is an unit vector from the spacecraft to the sun.

A.1.7 camera_position

typedef struct {**sim_time** *current_time*;
int *camera_type*;
int *take_picture*;
vector *bore_sight_INERTIAL*;
vector *orientation_INERTIAL*;
vector *bore_sight_COMET*;
vector *orientation_COMET*;} **camera_position**;

The **camera_position** data type contains a camera position in both INERTIAL and COMET-FIXED frames. The *current_time* field is the measurement time. The *camera_type* field indicates the camera type. If the *camera_type* field equals to 0, then it is a wide-field camera. If the *camera_type* field equals to 1, then it is a narrow-field camera. The *take_picture* field indicates whether to take a picture or not. If the *take_picture* field equals to 0, then don't take a picture. If the *take_picture* field equals to 1, then take a picture. The *bore_sight_** field is a vector from the spacecraft indicating the camera bore-sight. The *orientation_** field is an unit vector from the spacecraft indicating the camera orientation. The camera bore-sight vector is perpendicular to the camera orientation vector.

A.1.8 raw_image

typedef struct {**sim_time** *current_time*;
int *camera_type*;
int *image[512][512]*;} **raw_image**;

The **raw_image** data type contains a camera raw image. The *current_time* field is the time that the image is captured. The *camera_type* field indicates the camera type. If the *camera_type* field equals to 0, then it is a wide-field camera. If the *camera_type* field equals to 1, then it is a narrow-field camera. The *image* field contains the 512 by 512 pixel wide camera image. The *image* field is in gray scale, with black equals to 0 and white equals to 511. The index to the *image* field starts at 0 and ends at 511.

A.1.9 identified_image

typedef struct {**raw_image** *picture*;
    vector *trajectory_INERTIAL*;
    vector *thrust_INERTIAL*;
    vector *velocity_INERTIAL*;
    vector *deltaV_INERTIAL*;
    vector *sun_INERTIAL*;
    vector *bore_sight_INERTIAL*;
    vector *orientation_INERTIAL*;
    vector *trajectory_COMET*;
    vector *thrust_COMET*;
    vector *velocity_COMET*;
    vector *deltaV_COMET*;
    vector *sun_COMET*;
    vector *bore_sight_COMET*;
    vector *orientation_COMET*;} **identified_image**;

The **identified_image** data type contains an identified image in both the INERTIAL and COMET-FIXED frames. The *picture* field contains the raw image. The *trajectory_** field is a vector from the comet center to the spacecraft. The *thrust_** field is a vector from the spacecraft indicating the thrust. The *velocity_** field is a vector from the spacecraft indicating the velocity. The *deltaV_** is a vector from the spacecraft indicating the delta-V. The *sun_* field* is an unit vector from the spacecraft to the sun. The *bore_sight_** field is a vector from the spacecraft indicating the camera bore-sight. The *orientation_** field is an unit vector from the spacecraft indicating the camera orientation. The camera bore-sight vector is perpendicular to the camera orientation vector.

A.1.10 point

typedef struct {**int** *x*, *y*;} **point**;

The **point** data type contains a location on the 512 by 512 pixel wide image. If a **point** variable is equal to (0,0), then it is at the lower left corner of the image. If a **point** variable is equal to (0,511), then it is at the upper left corner of the image. If a **point** variable is equal to (511,0), then it is at the lower right corner of the image. If a **point** variable is equal to (511, 511), then it is at the upper right corner of the image. If a **point** variable is equal to (-1,-1), then it is not defined.

A.1.11 target

typedef struct {**identified_image** *data*;
    point *center*;
    point *upper_left*;
    point *lower_right*;
    int *target_type*;
    vector *displacement*;
    vector *target_COMET*;} **target**;

The **target** data type contains a target. The *data* field is the identified image from the camera. The *center* field is the center of the target on the image. The *upper_left* field is the upper left corner of the target on the image. The *lower_right* field is the lower right corner of the target on the image. The *target_type* field indicates the type of target. If the *target_type* field equals to -1, then it is not defined. If the *target_type* field equals to 1, then it is a sub-pixel level movement. If the *target_type* field equals to 2, then it is a pixel level movement. The *displacement* field indicates direction of the movement. The *target_COMET* field is the target vector in the COMET-FIXED frame.

A.1.12 global_parameters

typedef struct {**sim_time** *current_time*;
    vector *rotational_INERTIAL*;
    vector *rotational_COMET*;
    float *rotational_rate*;
    float *mass_density*;} **global_parameters**;

The **global_parameters** data type contains global parameters specifying the comet model. The *current_time* field indicates when these parameters are valid. The *rotational_** field is the unit vector indicating the rotational axis. The *rotational_rate* field is the rotational rate in degrees per second. The *mass_density* field is the mass density in kilograms per meter cube.

A.1.13 wire_frame

typedef struct {**sim_time** *current_time*;
    vector *coordinate_array[100]*;
    int *face_array[200]*;} **wire_frame**;

The **wire_frame** data type contains a wire-frame model of the comet. The *current_time* field indicates when the wire-frame model is valid. The *coordinate_array* field contains coordinate vectors of the wire-frame model. The coordinate vectors are indexed by their location in the *coordinate_array* field. The *face_array* field contains information about how coordinate vectors are connected. If the *face_array* field equals (0,65,87,-1, 64,91,56,-1,..,-1), then the coordinate vectors indexed by 0, 65, 87 form a face, -1 is a separator, coordinate vectors indexed by 64, 91, 56 form another face. The *face_array* field ends with a -1 separator.

A.2 Module Specification

The following is the message specification for each module.

A.2.1 Real-World Module

The Real-World module can receive the following broadcast messages.

TimeMsg "{sim_time}"
    Message to update the current time to "{sim_time}".
ResetMsg "{}"
    Message to reset the module.
ExitMsg "{}"
    Message to exit the module.

The Real-World module can receive the following multi-query message.

OkContinueMsg "{sim_time}" "{int}"
    Message to request permission to increment the current time by "{sim_time}". The Real-World module responds with 1 or 0 using "{int}".

The Real-World module can receive the following blocking command messages.

InjectCracksMsg "{}"

Message to inject cracks on the comet.
EjectParticlesMsg "{}"
Message to eject particles from the comet.
BreakCometMsg "{}"
Message to break the comet apart.
HardwareControlMsg "{hardware_control}"
Message to control the spacecraft hardware with "{hardware_control}".
NarrowControlMsg "{camera_position}"
Message to control the narrow-field camera with "{camera_position}".
DisplayTargetMsg "{target}"
Message to display the target, "{target}".
ImageProcessingOnMsg "{}"
Message to turn on image processing.
ImageProcessingOffMsg "{}"
Message to turn off image processing.

The Real-World module can send the following blocking command message.
NewImageMsg "{raw_image}"
Message to the Image-Identification module to process the new raw image, "{raw_image}".
NewLandmarkMsg "{landmark:10}"
Message to the Navigation module to process landmark measurements in {"landmark:10"}.
NarrowPositionMsg "{camera_position}"
Message to the Knowledge-Database module to update the narrow-field camera position with "{camera_position}".
WidePositionMsg "{camera_position}"
Message to the Knowledge-Database module to update the wide-field camera position with "{camera_position}".

A.2.2 Image-Identification Module

The Image-Identification module can receive the following broadcast messages.
TimeMsg "{sim_time}"
Message to update the current time to "{sim_time}".
ResetMsg "{}"
Message to reset the module.
ExitMsg "{}"
Message to exit the module.

The Image-Identification module can receive the following multi-query message.
OkContinueMsg "{sim_time}" "{int}"
Message to request permission to increment the current time by "{sim_time}". The Image-Identification module responds with 1 or 0 using "{int}".

The Image-Identification module can receive the following blocking command message.
NewImageMsg "{raw_image}"
Message to process the new raw image, "{raw_image}".

The Image-Identification module can send the following query messages.
CurrentTimeMsg "{}" "{sim_time}"
Message to query for the current time. The Simulation-Clock module responds with "{sim_time}".
NarrowEstimateMsg "{}" "{camera_position}"

Message to query for the current narrow-field camera position estimate. The Knowledge-Database module responds with "{camera_position}"
WideEstimateMsg "{}" "{camera_position}"
Message to query for the current wide-field camera position estimate. The Knowledge-Database module responds with "{camera_position}"
SpacecraftEstimateMsg "{}" "{spacecraft_position}"
Message to query for the current spacecraft position estimate. The Knowledge-Database module responds with "{spacecraft_position}"

The Image-Identification module can send the following blocking command messages.
ExecuteScienceMsg "{identified_image}"
Message to the Science module to process the "{identified_image}".
ExecuteAFASTMsg "{identified_image}"
Message to the AFAST module to process the "{identified_image}".

A.2.3 Science Module

The Science module can receive the following broadcast messages.
TimeMsg "{sim_time}"
Message to update the current time to "{sim_time}".
ResetMsg "{}"
Message to reset the module.
ExitMsg "{}"
Message to exit the module.

The Science module can receive the following multi-query message.
OkContinueMsg "{sim_time}" "{int}"
Message to request permission to increment the current time by "{sim_time}". The Science module responds with 1 or 0 using "{int}".

The Science module can receive the following blocking command message.
ExecuteScienceMsg "{identified_image}"
Message to process the "{identified_image}".

The Science module can send the following blocking query message.
CurrentTimeMsg "{}" "{sim_time}"
Message to query for the current time. The Simulation-Clock module responds with "{sim_time}".

The Science module can send the following blocking command message.
NewTargetMsg "{target}"
Message to the Knowledge-Database module to store the new target, "{target}".

A.2.4 AFAST module

The AFAST module can receive the following broadcast messages.
TimeMsg "{sim_time}"
Message to update the current time to "{sim_time}".
ResetMsg "{}"
Message to reset the module.
ExitMsg "{}"
Message to exit the module.

The AFAST module can receive the following multi-query message.
OkContinueMsg "{sim_time}" "{int}"
Message to request permission to increment the current time by "{sim_time}". The AFAST module responds with 1 or 0 using "{int}".

The AFAST module can receive the following blocking command message.
ExecuteAFASTMsg "{identified_image}"
Message to process the "{identified_image}".

The AFAST module can send the following blocking query messages.
CurrentTimeMsg "{}" "{sim_time}"
Message to query for the current time. The Simulation-Clock module responds with "{sim_time}".

The AFAST module can send the following blocking command message.
NewTargetMsg "{target}"
Message to the Knowledge-Database module to store the new target, "{target}".

A.2.5 Navigation Module

The Navigation module can receive the following broadcast messages.
TimeMsg "{sim_time}"
Message to update the current time to "{sim_time}".
ResetMsg "{}"
Message to reset the module.
ExitMsg "{}"
Message to exit the module.

The Navigation module can receive the following multi-query message.
OkContinueMsg "{sim_time}" "{int}"
Message to request permission to increment the current time by "{sim_time}". The Navigation module responds with 1 or 0 using "{int}".

The Navigation module can receive the following blocking command messages.
FlybyMsg "{vector}"
Message to perform a close flyby maneuver to "{vector}".
ToSafetyMsg "{}"
Message to perform an escape to safety maneuver.
OrbitingMsg "{}"
Message to perform an orbiting maneuver.
NewLandmarkMsg "{landmark:10}"
Message to process landmark measurements in {"landmark:10"}.

The Navigation module can send the following blocking query messages.
GlobalParametersMsg "{}" "{global_parameters}"
Message to query for the global parameters estimate. The Knowledge-Database module responds with "{global_parameters}".
WireFrameMsg "{}" "{wire_frame}"
Message to query for the wire-frame model estimate. The Knowledge-Database module responds with "{wire_frame}".

The Navigation module can send the following blocking commands.
SpacecraftProfileMsg "{spacecraft_position:10}"

Message to the Knowledge-Database module to update the spacecraft position profile with "{spacecraft_position:10}".
SpacecraftStateMsg "{int}"
Message to the Knowledge-Database module to update the spacecraft maneuvering state with "{int}".
HardwareControlMsg "{hardware_control}"
Message to the Real-World module to control the spacecraft hardware with "{hardware_control}".

A.2.6 Knowledge-Database Module

The Knowledge-Database module can receive the following broadcast messages.
TimeMsg "{sim_time}"
Message to update the current time to "{sim_time}".
ResetMsg "{}"
Message to reset the module.
ExitMsg "{}"
Message to exit the module.

The Knowledge-Database module can receive the following multi-query message.
OkContinueMsg "{sim_time}" "{int}"
Message to request permission to increment the current time by "{sim_time}". The Knowledge-Database module responds with 1 or 0 using "{int}".

The Knowledge-Database module can receive the following blocking query messages.
NarrowEstimateMsg "{}" "{camera_position}"
Message to query for the current narrow-field camera position estimate. The Knowledge-Database module responds with "{camera_position}"
WideEstimateMsg "{}" "{camera_position}"
Message to query for the current wide-field camera position estimate. The Knowledge-Database module responds with "{camera_position}"
SpacecraftEstimateMsg "{}" "{spacecraft_position}"
Message to query for the current spacecraft position estimate. The Knowledge-Database module responds with "{spacecraft_position}"
GlobalParametersMsg "{}" "{global_parameters}"
Message to query for the global parameters estimate. The Knowledge-Database module responds with "{global_parameters}".
WireFrameMsg "{}" "{wire_frame}"
Message to query for the wire-frame model estimate. The Knowledge-Database module responds with "{wire_frame}".
TargetsMsg "{}" "{target:10}"
Message to query for a list of 10 targets. The Knowledge-Database module responds with "{target:10}".
CurrentStateMsg "{}" "{int}"
Message to query for the current spacecraft state. The Knowledge-Database module responds with "{int}"

The Knowledge-Database module can receive the following blocking command messages.
NarrowPositionMsg "{camera_position}"
Message to update the narrow-field camera position with "{camera_position}".
WidePositionMsg "{camera_position}"
Message to update the wide-field camera position with "{camera_position}".
NewTargetMsg "{target}"

Message to store the new target, "{target}".

SpacecraftProfileMsg "{spacecraft_position:10}"
Message to update the spacecraft position profile with "{spacecraft_position:10}".

SpacecraftStateMsg "{int}"
Message to update the spacecraft maneuvering state with "{int}".

The Knowledge-Database module can send the following blocking command message.

TargetArrivedMsg "{target}"
Message to the Planner module to inform that the target, "{target}", has been discovered.

A.2.7 Planner Module

The Planner module can receive the following broadcast messages.

TimeMsg "{sim_time}"
Message to update the current time to "{sim_time}".

ResetMsg "{}"
Message to reset the module.

ExitMsg "{}"
Message to exit the module.

The Planner module can receive the following multi-query message.

OkContinueMsg "{sim_time}" "{int}"
Message to request permission to increment the current time by "{sim_time}". The Planner module responds with 1 or 0 using "{int}".

The Planner module can receive the following blocking command messages.

TargetArrivedMsg "{target}"
Message to inform that the target, "{target}", has been discovered.

PerformOrbitingMsg "{}"
Message to perform an orbiting maneuver.

PerformFlybyMsg "{target}"
Message to perform a close flyby maneuver.

PerformToSafetyMsg "{}"
Message to perform an escape to safety maneuver.

The Planner module can send the following blocking query messages.

TargetsMsg "{}" "{target:10}"
Message to query for a list of 10 targets. The Knowledge-Database module responds with "{target:10}".

CurrentStateMsg "{}" "{int}"
Message to query for the current spacecraft state. The Knowledge-Database module responds with "{int}"

The Planner module can send the following blocking command messages.

FlybyMsg "{vector}"
Message to perform a close flyby maneuver to "{vector}".

ToSafetyMsg "{}"
Message to perform an escape to safety maneuver.

OrbitingMsg "{}"
Message to perform an orbiting maneuver.

NarrowControlMsg "{camera_position}"
Message to control the narrow-field camera with "{camera_position}".

A.2.8 Simulation-Clock Module

The Simulation-Clock module can receive the following broadcast message.

ResetMsg "{}"
　　Message to reset the module.

ExitMsg "{}"
　　Message to exit the module.

The Simulation-Clock module can receive the following blocking command messages.

FastModeMsg "{}"
　　Message to change the simulation mode to FAST.

SlowModeMsg "{}"
　　Message to change the simulation mode to SLOW.

The Simulation-Clock module can send the following broadcast message.

TimeMsg "{sim_time}"
　　Message to update the current time to "{sim_time}".

The Simulation-Clock module can send the following multi-query message.

OkContinueMsg "{sim_time}" "{int}"
　　Message to request permission to increment the current time by "{sim_time}". Other module respond with 1 or 0 using "{int}".

A.2.9 User-Interface Module

The User-Interface module can send the following broadcast message.

ResetMsg "{}"
　　Message to reset the module.

ExitMsg "{}"
　　Message to exit the module.

The User-Interface module can send the following blocking command messages.

FastModeMsg "{}"
　　Message to the Simulation-Clock module to change the simulation mode to FAST.

SlowModeMsg "{}"
　　Message to the Simulation-Clock module to change the simulation mode to SLOW.

InjectCracksMsg "{}"
　　Message to the Real-World module to inject cracks on the comet.

EjectParticlesMsg "{}"
　　Message to the Real-World module to eject particles from the comet.

BreakCometMsg "{}"
　　Message to the Real-World module to break the comet apart.

ImageProcessingOnMsg "{}"
　　Message to the Real-World module to turn on image processing.

ImageProcessingOffMsg "{}"
　　Message to the Real-World module to turn off image processing.

PerformOrbitingMsg "{}"
　　Message to the Planner module to perform an orbiting maneuver.

PerformFlybyMsg "{target}"
　　Message to the Planner module to perform a close flyby maneuver.

PerformToSafetyMsg "{}"
　　Message to the Planner module to perform an escape to safety maneuver.

References

1. Albus, J. S. Outline for a theory of intelligence. IEEE Transactions on Systems, Man, and Cybernetics. 21(3):473-509; 1991.
2. Aljabri, A.; Eldred, D.; Goddard, R.; Gor, V. Kia, T.; Rokey, M.; Scheeres, D.; Wolff, P. Autonomous Serendipitous Science Acquisition for Planets (ASSAP). AIAA Paper 96-0699; 1996.
3. Brooks, R. A. A robust layered control system for a mobile robot. MIT A. I. Memo 864; 1985.
4. Chu, C.; Zhu, D. Q.; Udomkesmalee, S.; Pomerantz, M. I. Realization of autonomous image-based spacecraft pointing systems: planetary flyby example. SPIE's International Symposium on Optical Engineering in Aerospace Sensing, Space Guidance, Control, and Tracking Conference. Paper No. 2221-04; 1994.
5. Crippen, R. E. Measurement of subresolution terrain displacements using SPOT panchromatic imagery. Episodes. 15(1):56-61; 1992.
6. Ferrell, C. Robust agent control of an autonomous robot with many sensors and actuators. MIT A. I. Memo 1443; 1993.
7. Firby, R. J. Architecture, representation and integration: an example from robot navigation. Proceedings of the 1994 AAAI Fall Symposium Series Workshop on the Control of the Physical World by Intelligent Agents; 1994.
8. Firby, R. J. Adaptive execution in complex dynamic worlds. Ph.D. Thesis, Yale University Technical Report, YALEU/CSD/RR #672; 1989.
9. Hartley, R.; Pipitone, F. Experiments with the Subsumption architecture. Proceedings of the 1991 IEEE International Conference on Robotics and Automation. 1652-1658; 1991.
10. Pell, B.; Bernard D. E.; Chien, S. A.; Gat, E.; Muscettola, N.; Nayak, P. P.; Wagner, M. D.; Williams, B. C. A remote agent prototype for spacecraft autonomy. Proceedings of SPIE-96; 1996.
11. Scheeres, D. J. Close proximity and landing operations at small bodies. AIAA/AAS Astrodynamics Specialist Conference. AIAA Paper 96-3580; 1996.
12. Shih, J. Software architecture for autonomous spacecraft. 1995.
13. Simmons, R. G. Structured control for autonomous robots. IEEE Transactions on Robotics and Automation. 10(1): 34-43; 1994.
14. Simmons, R.; Goodwin, R.; Fedor, C.; Basista, J. Task control architecture programmer's guide to version 8.0. 1995.