COMPONENT-BASED SYSTEMS ENGINEERING FOR AUTONOMOUS SPACECRAFT

by

KATHRYN ANNE WEISS

B.S., Marquette University (2001)

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN AERONAUTICS AND ASTRONAUTICS

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
August, 2003
[September 2003]

Signature of Author [illegible]
Department of Aeronautics and Astronautics
August 2003

Certified by [illegible]
Professor Nancy G. Leveson
Department of Aeronautics and Astronautics
Thesis Supervisor

Accepted by [illegible]
Professor Edward Greitzer
H.N. Slater Professor of Aeronautics and Astronautics
Chair, Committee on Graduate Students

AERO

Component-Based Systems Engineering for Autonomous Spacecraft

by

Kathryn Anne Weiss

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the
Degree of Master of Science.

Abstract

The development of modern spacecraft is a challenging endeavor, especially in light of the increasing complexity of today's technology and ambitious mission goals, despite recent budget and personnel cutbacks. A new approach to spacecraft development that addresses many of the current issues facing the aerospace industry is described. The technique, called Component-Based Systems Engineering, is built upon a systems engineering development environment known as SpecTRM. An example of Component-Based Systems Engineering as applied to a series of autonomous spacecraft known as SPHERES is provided. Simulations of both one- and two-Sphere configurations are performed to illustrate not only the usefulness of the technique but also the benefits that Component-Based Systems Engineering provides.

Thesis Supervisor: Dr. Nancy G. Leveson
Professor of Aeronautics and Astronautics

Acknowledgements

I feel that writing this thesis has been a journey, through which I have learned a lot not only about engineering but also about myself. There have been many people who have helped make this journey a success and I would like to thank those individuals now.

First and foremost, I would like to thank my advisor Professor Nancy Leveson. You are one of the strongest and most respected women I know and an excellent role model. Thank you for all of your advice, guidance and especially for showing me how to be a strong woman in academia.

Thank you Professor Dave Miller and the SPHERES team, especially John Enright, for allowing me to work on your project and providing me with all of the information necessary to use SPHERES as my test-case. A big thanks goes to all of my labmates and Jeffrey and Patrick from Safeware for their constant support and help. I have learned so much from each of you and you have all, in your own ways, helped me to finish this thesis. Thank you very much.

To my best friends – Erin, Melissa and Sara – without you guys, I really think Boston and MIT would have driven me crazy! Thank you for listening to me all those times I called in the middle of the night freaking out about school. Thank you for keeping me grounded and sane through this whole process and for constantly reminding me of who I am and how far I have come.

Nicolas, thank you for being you. There aren't words to describe how much you have done for me over the past two years. You know that I could not have done this without you and I cannot thank you enough. You have been my anchor. Merci beaucoup!

Mom, Dad, Annie and Robert – I love you guys! Thank you for always being there for me and never letting me quit. You have carried me through thick and thin as I've made my way from college to graduate school. I can't tell you how much I appreciate your love and support.

I would like to thank God for giving me the strength to accomplish the endeavors that I thought were impossible. Finally, I would like to thank my grandma, Elsie Kirsch, for her courage and wisdom. She showed me that anything is possible and has made me believe that I can accomplish anything I set my mind to. She is my guardian angel and I would like to dedicate this thesis to her memory.

Table of Contents

List of Figures

Chapter 1
The Problem

In today's economy, NASA and its contractors do not have the political or economic backing to accomplish the highly publicized and successful space missions that, 50 years ago, received seemingly limitless funding and nationwide support. In an attempt to continue the exploration of space on budget allocations that seem to be waning with every year, the aerospace industry has been forced to rethink the way they engineer spacecraft. One way NASA has attempted to compensate for diminishing funds and support has been the Faster, Better, Cheaper approach. This chapter outlines the problems with Faster, Better, Cheaper and the inherent difficulties faced by the aerospace industry that set the stage for the approach to spacecraft engineering discussed in this thesis.

1.1 Faster, Better and Cheaper Spacecraft

Traditionally, spacecraft developed by NASA cost approximately $1 billion and take, on average, a decade to complete. The failure of one of these missions is debilitating to NASA and its contractors. In order to minimize the cost of mission failures while maximizing the amount of science done on a limited budget, NASA proposed a "Faster, Better, Cheaper," or FBC, approach to develop the next generation of space missions. During the 1990s, NASA operated under Chief Administrator Dan Goldin's FBC approach. These missions would have a budget of $150 to $350 million and take only three or four years to complete. NASA proposed accomplishing this goal through the use of small-scale Earth and space science missions, based upon proven technologies, which would theoretically require fewer managers and staff. Through reuse and a small managerial staff, NASA had hoped to launch ten or more spacecraft a year.

In 1999 alone, four missions (Mars Climate Orbiter, Mars Polar Lander, Wide Field Infrared Explorer and the two Deep Space 2 micro-probes) failed using the FBC approach and six of the 25 missions between 1996 and 2000 were lost [20]. Clearly, the FBC approach has not provided NASA with a spacecraft development approach that meets the new needs of creating spacecraft under a lower budget and in smaller time frame. There are many reasons why the

FBC approach has failed to fulfill its goals; resources were highly constrained and guidance was lacking due to many budget and workforce cuts throughout the 90s [18]. NASA accepted significant risk that manifested itself as a lack of attention to process [18]. One of the processes that suffered under this approach was software reuse. Because FBC decreased cost through the use of proven technologies, many aspects of spacecraft software were reused from one mission to the next. The previously stated lack of attention to process caused poor implementation of reuse. One example of the poor implementation of reuse is the loss of the Mars Climate Orbiter, described below.

1.1.1 Mars Climate Orbiter (MCO)

The Mars Climate Orbiter, or MCO, was part of the Mars Surveyor Program (MSP), which NASA established in 1994 to explore Mars. The first missions in MSP were the Mars Pathfinder and the Mars Global Surveyor. These missions were highly successful. The next two missions, Mars Polar Lander (MPL) and MCO, were scheduled to launch during the next minimum energy Earth-Mars transfer opportunity. The development teams had only 26 months to prepare for these two missions. In order to accomplish these demanding goals, the project decided to rely heavily on previous designs from MGS and Pathfinder.

The objective of the MCO mission was to deliver measurement devices for the collection of Martian climate and atmospheric data to a low, near-circular, Sun-synchronous orbit around Mars. The Mars Orbit Insertion (MOI) occurred on September 23, 1999. During MOI, the spacecraft signal is lost for nearly 25 minutes because of Mars occultation. The spacecraft signal dropped out 39 seconds earlier than predicted and never appeared again. From studying the telemetry, the investigation team was able to determine that there was an error in the spacecraft's navigation measurements of nearly 100 km. This resulted in a much lower altitude than expected and eventually led to the vehicle's break-up in the atmosphere.

An MGS-heritage Software Interface Specification indicated the exact format and units that the Angular Momentum Desaturation (AMD) files should have. This heritage specification indicated that Metric units should be used for the impuse-bit in the AMD files, because the equations supplied by the MGS-heritage software used Metric units. However, the equations in the AMD files that made the impulse-bit calculations were supplied by a vendor that used

English units. The conversion factor from English to Metric units was erroneously left out of the AMD files. Consequently, the AMD files did not conform to the heritage specification. The 4.45 conversion factor was left out of the MCO software, which led to an error in the state at closest approach to Mars and the break-up of MCO [5].

MCO provides a clear example of the difficulties with reuse. Not only did the MCO development team reuse software from MGS, but they also used vendor-supplied equations for in their AMD files. Because the MGS software had worked before with the vendor-supplied equations, there was little or no attention paid to the interface between these components. The conversion factor was included in the MGS software, but was never documented [5]. It is surprising that the MCO development team would not have checked that the units matched between the two software components. As stated in a paper written by several of the people involved in the development of the MCO and MPL, "the best chance to find and arrest this problem existed at the early levels of development" [5]. However, many of the procedures adopted during the early stages of development, such as the reuse of MGS-heritage software, were accepted because they were consistent with the FBC philosophy [5]. Reuse in and of itself did not cause the MCO to fail; the improper implementation of reuse that was caused by the shortened timeline and budgetary constraints was a major contributing cause to the loss.

Poor implementation of reuse has also occurred outside of the FBC approach. Similar problems also surfaced in a joint project between NASA and the European Space Agency (ESA) called SOHO and in the Ariane 5 launch vehicle. These aerospace accidents (SOHO and Ariane 5) and their relationship to reuse are described below to provide further insight into this difficult problem.

1.2 SOlar Heliospheric Observatory (SOHO)

SOHO, or the SOlar Heliospheric Observatory, is a joint effort between NASA and ESA to perform helioseismology and monitor the solar atmosphere, corona and wind. SOHO was launched on December 2, 1995, was declared fully operational in April of 1996, and completed a successful two-year primary mission in May of 1998. It then entered into its extended mission phase. After roughly two months of nominal activity, contact with SOHO was lost June 25,

1998. The loss was preceded by a routine calibration of the spacecraft's three roll gyroscopes (named A, B and C) and by a momentum management maneuver [21].

In order to increase the amount of science done during the mission and to increase the gyros' lifespans, a decision was made to compress the timeline of the operational procedures for momentum management, gyro calibration and science instrument calibration into one continuous sequence. The previous process had included a day between completing gyro calibration and beginning the momentum management procedures. Because the gyro calibration in the new compressed timeline was immediately followed by a momentum management procedure, despinning the gyros at the end of the gyro calibration and re-enabling the on-board software gyro control function was not required. However, after the gyro calibration, Gyro A was specifically despun in order to conserve its life, while Gyros B and C remained active. The modified predefined command sequence in the on-board control software had an error; it did not contain a necessary function to reactivate Gyro A, which was needed by the Emergency Sun Reacquisition. This omission resulted in the removal of the functionality of the spacecraft's normal safe mode, ESR, and ultimately caused the sequence of events that led to the loss of telemetry. In addition, there was another error in the software that resulted in leaving Gyro B in its high gain setting following the momentum management maneuver. This error originally triggered the ESR [21].

The first error was contained within a software function called A_CONFIG_N. ESR requires the use of Gyro A for roll control. Any procedure that spins down Gyro A must set a flag in the computer to respin Gyro A whenever the safe mode is triggered. When A_CONFIG_N was modified, the software enable command was omitted due to "a lack of system knowledge of the person who modified the procedure" [21]. Because the change had not been properly communicated, the operator procedures did not indicate that Gyro A had been spun down. In this accident, the two software errors were due to improper software change procedures due to lack of knowledge about the software and system design by those making the changes.

1.3 Ariane 5

On June 4, 1996, the maiden flight of the Ariane 5 rocket ended in disaster when, 40 seconds after launch, the launcher veered off its nominal flight path and exploded. The preliminary accident investigation showed that the launcher performed nominally until +36 seconds, at which point both the back-up and active Inertial Reference Systems failed, the two solid boosters swiveled to an extreme position causing the rocket to veer abruptly and finally the launcher self-destructed. The investigation board rapidly concluded that the Inertial Reference System (IRS) was at the heart of the accident.

Because the design of the Ariane 5 IRS was similar to the one used on the Ariane 4, a decision was made to reuse the IRS software from the Ariane 4 on the Ariane 5. The time sequence of the Ariane 5 lift-off is significantly different from that of the Ariane 4. An alignment function included in earlier versions of the rocket to restart an aborted countdown could no longer be used in Ariane 5. However, the function was left in the Ariane 5 software for commonality reasons, "based on the view that, unless proven necessary, it was not wise to make changes in software which worked well on Ariane 4" [15]. The alignment function had not been shut down in the previous Ariane rockets until 50 seconds into flight mode. Therefore, the unchanged alignment function would also remain active 50 seconds into the flight mode of the Ariane 5. In addition, the trajectory of Ariane 5 also differs from that of Ariane 4 and results in considerably higher horizontal velocity values. The higher horizontal velocity led to a BH (horizontal bias variable) value that was much higher than expected. This, in turn, caused an operand error in an alignment function. An exception was raised causing the nozzle of the solid rocket boosters to deflect, from which the launcher experienced high aerodynamic loads that led to its explosion 39 seconds into flight [15]. Clearly, the reused Ariane 4 software was not suitable for the Ariane 5 without considerable changes.

In an analysis of reuse and the Ariane 5 accident, Weyuker observes that many engineers believe if the software components are reusable, they do not have to be reevaluated for integration into the new system [27]. The Ariane 5 and MCO accidents were examples of what happens when software components are not reevaluated and properly integrated into the new system. The poor implementation of reuse led to complete losses in both cases.

The software components Weyuker refers to are the building blocks of an approach to software development called Component-Based Software Engineering, or CBSE. Component-Based Software Engineering is one way the software industry has incorporated reuse into its software development practices. The next chapter defines Component-Based Software Engineering and discusses its costs (why it is easy to poorly implement CBSE) and benefits. The third aerospace accident example involved an improper implementation of software change, a topic that will be addressed in Chapter 3. Unfortunately, poor reuse practices are not the only problems currently facing the aerospace industry. The next section describes other factors that are making the development of the next generation of space vehicles an even more difficult task.

1.4 Inherent Difficulties of the Aerospace Industry

Not only are aerospace companies faced with the problems caused by poor implementation of component-based software engineering and reuse, but they are also facing difficulties within their own industry. The following problems in the aerospace industry add to the complexity of developing a suitable methodology for creating the next generation of spacecraft.

1.4.1 Spacecraft Software Structure and a Lack of Autonomy

Traditionally, spacecraft software has been highly event-based: a time or an action triggers another action in software. Consequently, the spacecraft must adhere closely to its predefined operational model to assure that mission objectives are achieved. These sequences are extremely specific to each spacecraft and mission. Therefore, much of the control software cannot be reused from one spacecraft or mission to another.

In addition, the occurrence of unpredictable events outside nominal variations is dealt with by high-level fault protection software. This software may be inadequate if time or resources are constrained and recovery actions interfere with satisfying mission objectives [24]. In this case, the spacecraft enters a safe mode in which all systems are shut down except for those needed to communicate with Earth. The spacecraft then waits for instructions from the ground controller [6]. Safe mode is problematic for two reasons. First, if the spacecraft is far from earth, there is a large communication delay. During the cruise phase of flight, the delay may not cause any problems; the spacecraft has time to await new instructions from Earth. However, if the spacecraft is executing an event sequence during a safety-critical flight phase, such as an orbit insertion, the procedure may prove to be extremely costly and possibly fatal [6]. Second, large and expensive deep-space mission such as Cassini require an enormous number of ground controllers to support each phase of flight.

Recently, researchers at the Jet Propulsion Laboratories (JPL) have proposed a new approach to designing spacecraft software. The Mission Data System, or MDS, is presently under development by NASA and is based on the principles of Artificial Intelligence (AI). MDS is a goal-based system, which is defined as a system in which all actions are directed by goals instead of commands [24]. A goal is a constraint on a state over a time interval and elaboration is the process by which a set of rules recursively expands a high level goal into a goal network [6]. The new low-level goals are then merged with the previous goals, thereby providing the spacecraft with the ability to deal with previously unknown situations. The summation of these goals forms the knowledge base from which the agents in the AI architecture will gather information to make decisions for the spacecraft.

The proponents of MDS suggest that the problems with traditional software (specificity and inability to handle faults) are easily dealt with through the use of such an architecture. First, the entire architecture is reused from one spacecraft to the next; the only aspect of MDS that changes are the goals for a particular mission. Second, fault tolerance is no longer considered a separate entity. Because a failure mode or other anomalous condition is treated as just another possible set of system states, the spacecraft does not have to alter its nominal operations [6]. It simply relegates the fault by attempting to fulfill its mission goals given its current state, a process no different than if the spacecraft was not in a fault detected state.

Although MDS appears to be a suitable alternative to current spacecraft software architecture, several issues have been raised that question the approach. First, some researchers argue that AI has not matured to the point where it can be reliably used in safety-critical systems. Second, MDS has been under development for nearly ten years, and it is still not finished. Although JPL touts Remote Agent as a complete success for AI, Remote Agent never actually controlled Deep Space 1. There has been no proof from either MDS or the Remote Agent project that suggests that AI is a feasible and appropriate software architecture for a spacecraft controller. Finally, is there a middle ground between time-stamped commands and artificial intelligence? These two approaches seem to lie on the opposite extremes of the software architecture spectrum. A more moderate approach may be more suitable for the next generation of spacecraft.

1.4.2 Loss of Domain Knowledge

As the Apollo-era spacecraft engineers retire, the wealth of knowledge that they have acquired throughout their careers has the potential for being lost [12]. The knowledge needs to be captured and recorded in an easily readable format so that it can be passed on to the next generation of spacecraft engineers. One of the most important aspects of this domain knowledge is rationale, for example, why a certain design or implementation decision was made. In the past, this information has not been transferred and takes years of experience to learn. This is the type of information that needs to be recorded.

1.4.3 Miscommunication Among Multi-disciplinary Engineering Teams

Multi-disciplinary engineering teams are common in the aerospace industry. Spacecraft need a variety of subsystems that range from attitude determination and control, to communications and power. These subsystems are all controlled by software that allows the spacecraft to accomplish its mission objectives. Spacecraft development requires the expertise of engineers from fields as different as mechanical engineering and computer science. These individuals have extremely diverse backgrounds, talents and communication skills. They use different terminology (sometimes for the same concepts) and language specific to their field. They are accustomed to certain tools and problem solving strategies. These differences make the engineering effort difficult and create communication problems among team members. A common medium is needed for cross-disciplinary communication on spacecraft engineering teams to help facilitate understanding among team members and decrease the ambiguity caused by their diverse backgrounds.

1.5 Summary

There are two main problem areas facing the aerospace industry today. First, the poor implementation of component-based software engineering practices, wide spread code reuse and changes to existing software are creating catastrophic failures for many companies in the industry. These losses cannot continue to be sustained especially in light of the push to create faster, better and cheaper spacecraft. Second, there are problems within the aerospace industry itself. The event-based software sequences that have been used by the spacecraft industry for years are becoming obsolete. New, more complex missions are forcing engineers to rethink the way they design spacecraft software. The lack of spacecraft autonomy forces aerospace companies to maintain a large number of expensive ground controllers for fault tolerance. The attempts to alleviate these software structure problems with artificial intelligence have not yet come to fruition. The aging of the intellectual workforce at NASA and its contractors threatens to eliminate a large source of domain knowledge that has been acquired through years of experience. And, finally, communication between engineers from various fields may lead to severe misunderstandings, which can eventually lead to costly mishaps.

1.6 Thesis Outline

This thesis proposes a new approach to spacecraft development that uses the principles of Component-Based Software Engineering and Systems Engineering to provide engineers with an environment in which it is feasible to produce spacecraft faster under tighter budgets. Chapter 2 provides background on Component-Based Software Engineering and Systems Engineering. It then defines the approach to spacecraft development that combines the principles of these two techniques and which can be used to solve some of the problems outlined in this chapter. This technique is known as Component-Based Systems Engineering.

Chapter 3 identifies and describes the systems engineering development environment called SpecTRM, which provides a platform for Component-Based Systems Engineering. SpecTRM is a toolkit that allows users to create intent specifications, which will serve as the reusable components that are central to this approach.

Chapter 4 applies Component-Based Systems Engineering to SPHERES, a group of satellites that perform high-risk metrology, control and autonomy algorithms inside the United States Node of the International Space Stations. Two Guest Scientist Programs are modeled to illustrate the ease with which additional spheres can be added to the system after intent specifications have been created. Finally, Chapter 5 contains the conclusions drawn from the research and the test case.

Chapter 2
Component-Based Systems Engineering

This chapter defines Component-Based Software Engineering and the current uses of the technique in addition to some of its negative aspects. Various definitions of systems engineering are also provided as well as identification of a few characteristics of systems engineering that are constant throughout these definitions. A methodology for spacecraft development is then outlined that combines the principles of both Component-Based Software Engineering as well as systems engineering

2.1 Component-Based Software Engineering and Reuse

Component-Based Software Engineering, or CBSE, is the process by which software code is specified, designed, implemented, tested and maintained using a collection of functional elements that communicate through pre-specified interfaces. It has been proposed that reusing these components can significantly lower development costs and shorten development cycles. It can also lead to software systems that require less time to specify, design, test and maintain, while satisfying high reliability requirements [26].

Many software engineers suggest that creating the components with a domain-specific language can further enhance the benefits of CBSE. Domain-specific languages (DSLs) are created to be as close as possible to the expert's conceptual view of the application domain, thereby allowing the user to easily describe their systems [23]. When these DSLs used in combination with CBSE are high-level, the approach is referred to as Domain-Specific Software Architectures (DSSAs) [1]. Many industries, including aerospace, are researching the usefulness of DSSAs and CBSE for software development. At Honeywell, research is being conducted in applying DSSAs to Guidance, Navigation and Control [10]. The researchers suggest that using DSSAs helps software developers experience the benefits of CBSE and the benefits of DSLs. Using DSSAs allows multiple phases of the lifecycle (design, implementation and testing) to be reused. This is an improvement over CBSE, because it only reuses code, which comprises merely 10-20% of the software development effort [10].

However, many software professionals agree that a practical approach to performing any of these component-based software development techniques has not yet been developed [1]. The lack of successful implementations of these techniques is due to a variety of industrial factors as well as some of the observed limitations of CBSE described below.

First, software components that have been successful within a different project are assumed to be "proven" and do not need to be reevaluated for integration into the new system, as seen in both the Ariane 5 and MCO accidents. In many cases, when the engineers do want to properly integrate the component into their system's software, sufficient documentation does not accompany the code, due to proprietary reasons on the part of the vendor. Consequently, engineers have a fairly difficult time determining how the module actually works and how it will operate within its new environment. Third, unused portions of a reusable component are often left in the system, because the developers do not know the effects of removing code snippets. Finally, reusable components are difficult to change correctly because of the aforementioned lack of documentation.

Clearly, reuse has both its benefits and its costs. However, if reuse is to really contribute to faster, better and cheaper spacecraft, the reuse methods NASA and its contractors employ need to be refined. The rest of this chapter defines systems engineering and describes how its principles can be applied to CBSE to help the aerospace industry implement reuse practices more effectively.

2.2 Systems Engineering

A system can be defined as a series of interrelated components that work together toward a common p urpose. T he s ystems d eveloped t oday a re b ecoming i ncreasingly c omplex. T his complexity is a result of interconnections, interactions and interdependencies between the components (which may be systems themselves) that comprise the system. Because of this complexity, one component of the system cannot be engineered independently of the other components. A system-wide view of the components and how they contribute to and interact with the system-as-a-whole must be taken. The components must be engineered within the context of their place within the system. This approach is called systems engineering.

There are many definitions of systems engineering. The NASA Systems Engineering Handbook defines systems engineering as "a robust approach to the design, creation and operation of systems" [19]. This approach consists of identifying and quantifying system goals, creating alternative design concepts, performing a trade-off analysis of these designs, selecting and implementing the best design, verifying that the design was properly built and implemented and finally assessing how well the system meets the goals [19].

Another definition comes from the International Council on Systems Engineering: "Systems engineering is an interdisciplinary approach and means to enable the realization of successful systems. It focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, then proceeding with design synthesis and system validation while considering the complete problem:

- Operations
- Performance
- Test
- Manufacturing
- Cost & Schedule
- Training & Support
- Disposal

Systems engineering integrates all the disciplines and specialty groups into a team effort forming a structured development process that proceeds from concept to production to operation" [11].

The MIT's Engineering Systems Division characterizes systems engineering as "a process for designing systems that begins with requirements, that uses and/or modifies an architecture, accomplishes function and/or physical decomposition, and accounts for the achievement of the requirements by assigning them to entities and maintaining oversight on the design and integration of those entities" [16]

There are many more definitions of systems engineering, but these definitions and the examples above all contain many similar ideas. First, systems engineering is multidisciplinary. Complex systems are built up of a variety of subsystems, which include everything from human operations to electronics. The systems engineering effort involves using the expertise of the engineers from each of the engineering disciplines that contribute to the system functionality. Systems engineering is also process inclusive. It involves interdisciplinary trade-off analyses and evaluation of customer need and requirements satisfaction at each stage in the development lifecycle.

Systems Engineering is a much superior approach to the development of today's complex systems, because it requires that engineers have a thorough understanding of not only the subsystem with which they are working, but also how the subsystem affects its environment. Caldwell and Chau note that, "Each person must be able to see a larger portion of the whole than the traditional partitioning according to subsystems" [2]. In particular, an avionics engineer must not only understand all of avionics but also their many interactions with other parts of the space system [2]. In addition, systems engineering allows engineers to reduce cost and risk through the evaluation of multiple design alternatives at each stage in the development lifecycle [19]. Clearly, systems engineering must play an integral role in a new approach to engineering the next generation of spacecraft.

2.3 What is Component-Based System Engineering?

As discussed in the previous chapter, reuse is critical in decreasing the cost of spacecraft development. However, when Component-Based Software Engineering is implemented incorrectly, its effectiveness is decreased and in some cases catastrophic accidents occur. The Mars Climate Orbiter and Ariane 5 accidents are examples of what happens when CBSE and reuse are implemented improperly. Furthermore, the SOHO accident exemplifies poor implementation of software change. Systems engineering was created to deal with these types of problems. Systems engineering stresses that subsystems must be developed as parts of a whole instead of independent entities. The awareness of interconnections, interactions and interdependencies helps to prevent accidents that stem from poor reuse and change practices. Since CBSE focuses on reusing individual software components alone without addressing its implications on other aspects of the system, this awareness is not present and accidents like MCO, Ariane 5 and SOHO can occur. Combining the ideas and processes of systems engineering with CBSE may alleviate some of these problems.

In a combined technology, engineering teams would perform Component-Based *Systems* Engineering instead of Component-Based *Software* Engineering, in which each component or subsystem is developed using a systems engineering development environment. As previously stated, systems engineering is a process-inclusive approach to designing a system, that is, systems engineering development environments are created to foster documentation, trade-off analysis and testing throughout every stage of the engineering development lifecycle.

A systems engineering approach is especially important for the software portions of the system. Today's systems are primarily software driven. Accidents involving computers are usually the result of flaws in the software requirements, not coding errors [13]. Although the system is developed with a component-based approach, applying systems engineering principles to this approach helps engineers to recognize software interconnections at every stage of the lifecycle, including requirements specification. Thorough documentation, trade-off analyses, testing and the recognition that each software component is a part of the system-as-a-whole increases the quality of requirements specifications by uncovering problems early in the lifecycle and thereby decreasing the cost of correcting these mistakes. Requirements specifications are analyzed before any code is ever written or any hardware implementation is completed.

The first step in applying the Component-Based Systems Engineering approach involves a decomposition of the system. Depending on the properties of the system, this may be a functional, physical or logical decomposition. Functional decomposition is a natural approach for spacecraft, which are composed of components that are grouped into subsystems based on the functionality they provide to the system-as-a-whole. At the highest level, software manages the spacecraft and is called the controller. The controller integrates the various subsystems and usually allocates resources and tasks to them. The next level of decomposition is the subsystem level. All spacecraft have similar subsystems, which include attitude determination and control, power, thermal, guidance and navigation, communications and propulsion. These subsystems are directed by the spacecraft controller to accomplish mission objectives. The subsystems can be further decomposed into their constituent components. For example, most attitude determination and control subsystems are composed of some combination of the following components: reaction control systems, reaction wheel assemblies, inertial measuring units, star trackers, sun sensors and horizon sensors. The spacecraft decomposition is complete when the individual hardware elements are reached. For the generic spacecraft, functional decomposition yields the three levels discussed above: the spacecraft controller, the subsystems and the individual components. Figure 1 illustrates an example decomposition of a generic spacecraft.

Now that the spacecraft has been decomposed into a series of components, the next step in Component-Based Systems Engineering is to develop the spacecraft from the individual components to the spacecraft controller using a systems engineering development environment. Because the spacecraft developed by a given company follow many common development patterns and use similar components, spacecraft components could be built to be generic, which makes them reusable. These components would be created with the previously described systems engineering approach in the systems engineering development environment and therefore the entire engineering development process can be reused instead of merely software code. The company would create a library of these components. When a new spacecraft is needed, components are combined to create a generic subsystem. The subsystems are refined to reflect the specific goals and mission objectives of the project. They are then combined to form the spacecraft controller for the particular spacecraft.

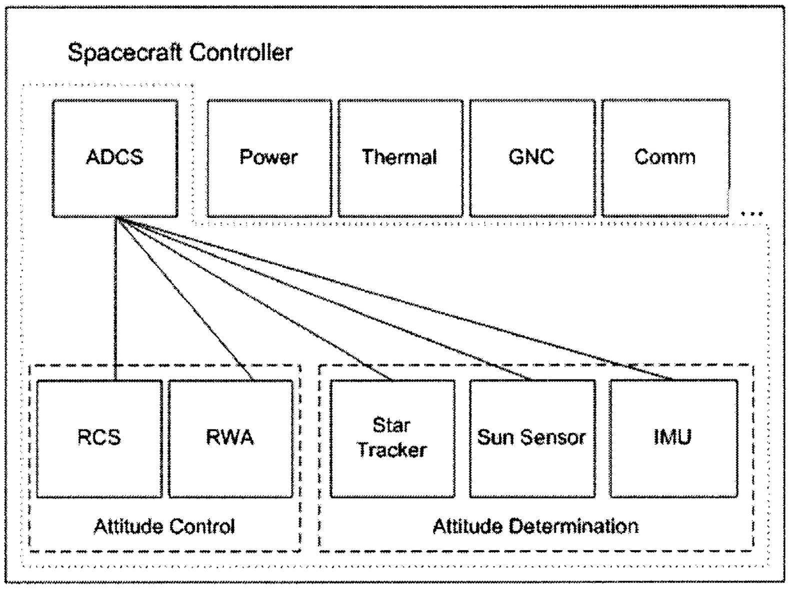


Figure 1. Example Spacecraft Decomposition

This methodology decreases the time it takes to develop a new spacecraft because it does not start from scratch, one of the main advantages of using a component-based development approach. Because spacecraft engineers develop the components, they are specific to the domain of spacecraft engineering, which aids engineers on other projects to easily incorporate these components into their projects. In addition, the entire process of developing the components and subsystems of the spacecraft with the principles of systems engineering and the assembly of the subsystems and controller is reused. Details specific to the spacecraft and its mission can be easily added to the generic components, making the design suitable for the wide range of spacecraft applications.

2.4 Summary

Combining the systems engineering and component-based approaches to project development allows engineers to experience the benefits of Component-Based Software Engineering without the detrimental effects of improper implementation of reuse. Instead of performing CBSE, engineers can perform Component-Based *Systems* Engineering, in which the entire process of developing a component or subsystem of a system is reused. The development is performed in a systems engineering development environment, which supports the principles of systems engineering such as a common means of communication between the various types of engineers on the development team as well as placing the component or subsystem in context within the larger system. The next chapter describes a systems engineering development environment that provides the foundation for and implementation of Component-Based Systems Engineering.

Chapter 3
SpecTRM-GSC

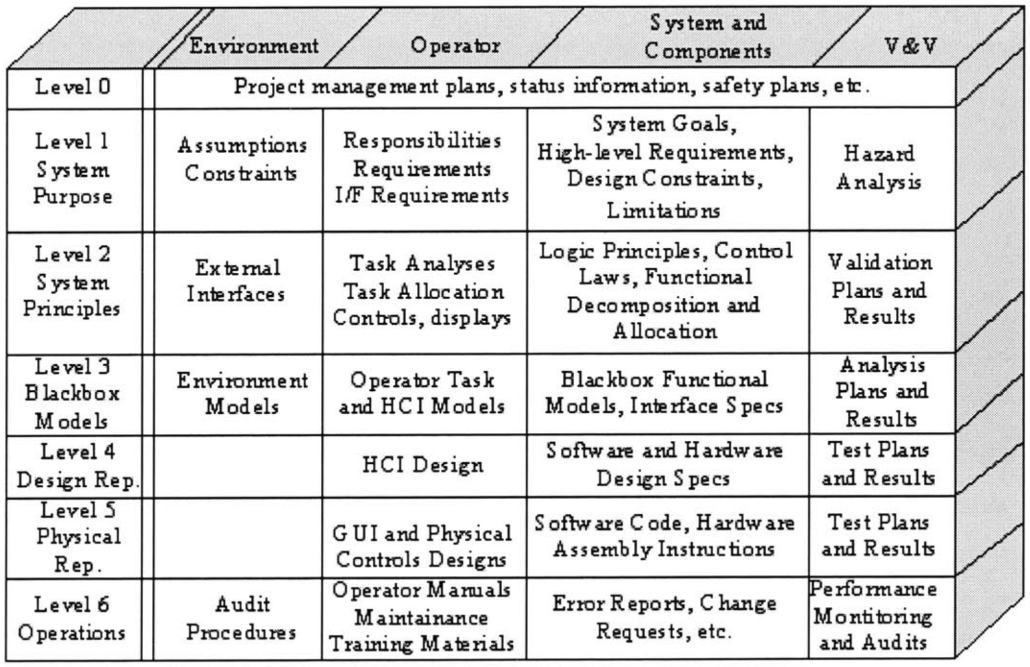
The construction of reusable components using a systems engineering approach requires a platform upon which these components can be built. This chapter proposes the use of intent specifications and SpecTRM as a platform for Component-Based Systems Engineering. Each component is an intent specification built in SpecTRM, which is a toolkit that allows users to create intent specifications as well as perform formal analyses on the model. Finally, the properties of the reusable components that emerge from the use of the Component-Based Systems Engineering technique and SpecTRM are identified and described.

3.1 Intent Specifications and SpecTRM

Intent specifications are based on research in human problem solving and on basic principles of system theory. An intent specification differs from a standard specification primarily in its structure: the specification is structured as a set of models designed to describe the system from different viewpoints, with complete traceability between the models. The structure is designed (1) to facilitate the tracing of system-level requirements and design constraints down into detailed design and implementation, (2) to assist in the assurance of various system properties (such as safety) in the initial design and implementation, and (3) to reduce the costs of implementing changes and reanalysis when the system is changed, as it inevitably will be. Because of its basis in research on how to enhance human problem solving, intent specifications should enhance human processing and use of specifications and our ability to perform system design and evolution activities. Note that no extra specification is involved (assuming that projects produce the usual specifications), but simply a different structuring and linking of the information so that specifications provide more assistance in the development and evolution process [14].

There are seven levels in an intent specification as seen in Figure 1 [14]. Levels do not represent refinement, as in other more common hierarchical structures, but instead each level of an intent specification represents a completely different model of the same system and supports a different type of reasoning about it: each model or level presents a complete view of the system, but from a different perspective. The model at each level may be described in terms of a different set of attributes or language. Refinement and decomposition occurs within each level of the specification, rather than between levels [14].

The top level (Level 0) provides a project management view and insight into the relationship between the plans and project development. Level 1 of an intent specification is the customer view and assists system engineers and customers in agreeing on what should be built and whether that has been accomplished. It includes system goals, high-level requirements, design constraints, hazards, environmental assumptions, and system limitations. The second level, System Design Principles, is the system engineering level and allows engineers to reason about the system in terms of the physical principles and laws upon which the system design is based.



	Environment	Operator	System and Components	V&V
Level 0	Project management plans, status information, safety plans, etc.			
Level 1 System Purpose	Assumptions Constraints	Responsibilities Requirements I/F Requirements	System Goals, High-level Requirements, Design Constraints, Limitations	Hazard Analysis
Level 2 System Principles	External Interfaces	Task Analyses Task Allocation Controls, displays	Logic Principles, Control Laws, Functional Decomposition and Allocation	Validation Plans and Results
Level 3 Blackbox Models	Environment Models	Operator Task and HCI Models	Blackbox Functional Models, Interface Specs	Analysis Plans and Results
Level 4 Design Rep.		HCI Design	Software and Hardware Design Specs	Test Plans and Results
Level 5 Physical Rep.		GUI and Physical Controls Designs	Software Code, Hardware Assembly Instructions	Test Plans and Results
Level 6 Operations	Audit Procedures	Operator Manuals Maintainance Training Materials	Error Reports, Change Requests, etc.	Performance Montitoring and Audits

Figure 2. Intent Specification Hierarchy

The third, or Blackbox Behavior level, enhances reasoning about the logical design of the system as a whole and the interactions between the components as well as the functional state without being distracted by implementation issues. This level acts as an unambiguous interface between systems engineering and component engineering to assist in communication and review of component blackbox behavioral requirements and to reason about the combined behaviour of individual components using informal review, formal analysis, and simulation. The language used on this level, SpecTRM-RL, has a formal foundation so it can be executed and subjected to formal analysis while still being readable with minimal training and expertise in discrete math.

The next two levels provide the information necessary to reason about individual component design and implementation issues. Finally, the sixth level provides a view of the operational system. Each level is mapped to the levels above and below it. These mappings provide the relational information that allows reasoning across the hierarchical levels and tracing from high-level requirements down to implementation and vice versa.

Intent information represents the design rationale upon which the specification is based. This design rationale is integrated directly into the specification. Each level also contains information about underlying assumptions upon which the design and validation is based. Assumptions are especially important in operational safety analyses. When conditions change such that the assumptions are no longer true, then a new safety analysis should be triggered. These assumptions may be included in a safety analysis document (or at least should be), but are not usually traced to the parts of the implementation they affect. Thus even if the system safety engineer knows that a safety analysis assumption has changed (e.g., pacemakers are now being used on children rather than the adults for which the device was originally designed and validated), it is a very difficult and resource-intensive process to determine which parts of the design used that assumption [14].

The safety information system or database is often separated from the development database and specifications. In the worst case, system and software safety engineers carefully perform analyses that have no effect on the system design because the information is not contained within the decision-making environment of the design engineers and they do not have access to it during system design. By the time they get the information (usually in the form of a critique of the design late in the development process), it is often ignored or argued away because changing the design at that time is too costly. Intent specifications integrate the safety database and information into the development specifications and database so that the information needed by engineers to make appropriate tradeoffs and design decisions is readily available [14].

Interface specifications and specification of important aspects of environmental components are also integrated into the intent specification, as are human factors and human interface design. The separation of human-automation interface design from the main system and component design can lead to serious deficiencies in each. Finally, each level of the intent specification includes a specification of the requirements and results of verification and validation activities of the information at that specification level [14].

SpecTRM, which stands for Specification Toolkit and Requirements Methodology, is a development environment that allows users to easily create, modify and analyze intent specifications. SpecTRM includes many features important to the intent specification process. First, an empty intent specification in SpecTRM contains headings that, when filled out, help ensure specification completeness. By providing the user with an initial structure to their specification, SpecTRM helps the user to think about aspects they may have otherwise left out. Second, SpecTRM provides an easy link creator. Links between levels provide traceability within the specification from the highest requirements all the way down to implementation. This is especially useful for tracking changes and performing interface testing. Finally, SpecTRM provides various analyses that can be performed on the Level 3 blackbox model [25].

3.1.1 Analyses

SpecTRM currently provides two analyses that can be performed on the individual intent specifications: non-determinism and robustness. A model is deterministic if for any given system state and set of inputs, there is only one transition for each state and mode [25]. A model is robust, if for any given system state and set of inputs, a transition exists, i.e. a behavior is defined for all possible inputs [25]. These analyses allow the system engineers to eliminate all inconsistencies and incompleteness before the simulation is run. The Level 3 models can be checked automatically for these properties [7].

3.1.2 Simulations

SpecTRM models are executable. Because the Level 3 blackbox model is a formal representation of an underlying state machine, the model can be executed given a set of inputs.

Individual models can be executed in isolation and multiple models can be executed in an environment in which they interact with each other. Components can be linked to their parent subsystems and the subsystems to the controller to simulate the system-as-a-whole.

There are many benefits to running simulations in SpecTRM. First, simulations allow system developers to observe the results of interactions between components and the functionality of the subsystem specification and model. Testing blackbox behavior is especially important at this stage in the development lifecycle, because errors in the requirements specifications and/or the blackbox model can be uncovered before any code has been implemented. After the spacecraft has been created and deployed, changes will need to be made to the on-board software. Code maintenance comprises nearly 70% of the software lifecycle and changes to the software can be costly [26]. An executable state machine provides the software maintainers with the ability to incorporate changes to the code from the formal requirements specification, and simulate the effects those changes will have on the rest of the system, again before any code has been implemented. As evidenced by the SOHO accident described in Chapter 1, improper software change procedures can be as detrimental to a system as poor implementation of reuse.

Third, executable blackbox models help developers to perform trade-off analyses. Engineers can simulate alternative design strategies and determine which approach is most suitable given the constraints and requirements of the system. Finally, different types of visualizations of the underlying state machine allow users to facilitate the creation of a mental model of the system's functioning. A high quality mental model of the system will improve the requirements creation and reviewing process [3]. Clearly, having a formal, executable, blackbox model of the system provides engineers with the variety of benefits that aid in the proper implementation of a component-based development approach.

3.2 SpecTRM-GSC

Intent specifications provide the features needed to perform Component-Based Systems Engineering. As described in Chapter 2, the construction of a system using Component-Based Systems Engineering begins with a decomposition of the system. After the system has been decomposed into its subsequent subsystems and individual components, the system is developed from the bottom-up. In the example given in this thesis, the component intent specifications were constructed using SpecTRM. For spacecraft, the intent specifications were labeled SpecTRM-GSCs, or SpecTRM-Generic Spacecraft Components. There are four main characteristics of SpecTRM-GSCs that are crucial to the success of this technique: each component must be fully encapsulated, have well-defined interfaces, be reusable and contain component-level fault protection.

3.2.1 Fully Encapsulated

Each SpecTRM-GSC must be fully encapsulated, meaning that all the functionality of that component should be contained within the component intent specification. Conventionally, much of the control and software for attitude determination and control components was traditionally distributed between the controller, the subsystem and component itself. By fully encapsulating the operations of each device within one intent specification, the modularity of design process and ease with which components are reused increases.

For example, a reaction wheel assembly will always receive torque commands as inputs. It then spins the reaction wheels to achieve the desired torque. Instead of associating these operations with the attitude determination and control subsystem (ADCS) from the beginning of development, an intent specification for the reaction wheel assembly is written independently of the ADCS, thereby disassociating the component from its possible uses. By capturing only what and how the reaction wheel assembly provides instead of what it will be used for, the component becomes far more modular not only between spacecraft but also within the same spacecraft. In other words, the component can be used in the ADCS of many different spacecraft as well as different subsystems of the same spacecraft. For example, in some extreme cases, thermal subsystems can use the friction of the spinning reaction wheels to generate heat.

3.2.2 Well-Defined Interfaces

Like components in Component-Based Software Engineering, the SpecTRM-GSCs must also have well-defined interfaces. All components must adhere to standard naming conventions and input/output requirements. It is extremely important that the construction of the Level 3 blackbox models of the SpecTRM-GSCs follow a consistent pattern in terms of how blackbox elements are named in order to avoid confusion and increase the ease with which components are combined to create subsystems. Consistency in the construction of the intent specifications is especially important in the case of inputs and outputs, because these are the modeling elements through which the components communicate with their parent subsystem. Therefore, a convention for both names and the type of information that the inputs and outputs transfer must be defined from the beginning of development and then strictly followed.

3.2.3 Generic

Because the components are fully encapsulated and have well-defined interfaces, they are also reusable. To enhance this reusability, specific information should be left out of the specification, making each SpecTRM-GSC highly generic. The system engineer inserts system-specific information when the components are used for a particular spacecraft subsystem. For example, when a digital sun sensor is used in an ADCS specification, the system engineer must specify the particular model being used and other information specific to that model number. If the sun sensor selected is the Adcole Digital Sun Sensor Model 18960, for example, the engineer must specify that the sun sensor uses a 15 bit input from its sensor heads. In the SpecTRM-GSCs, the use of both bold face and underlining highlights such information. These font characteristics alert the system engineer that the information is specific to a particular model number and should be changed when the generic component is instantiated in a particular spacecraft design.

3.2.4 Component-Level Fault Protection

Component-Based Systems Engineering employs three levels of fault-protection: intra-component fault protection, inter-component fault protection and inter-subsystem fault protection. These three levels ensure that fault protection covers the entire system; not only must the design account for component failures, but also for failures resulting from the interactions between components and subsystems. At the intra-component level, the fault protection logic assures that if the component is working in an off-nominal mode, it will alert its subsystem.

Then, at the inter-component level, the subsystem determines how to handle that fault. This feature is especially important in autonomous spacecraft.

3.3 Summary

Intent specifications and SpecTRM provide a platform upon which Component-Based System Engineering can be performed. SpecTRM-GSCs help to capture domain knowledge through recording the rationale behind decision-making at each step in the development lifecycle. They abstract away the details of design so that the specifications can be reusable from one project to the next. Various analyses can be performed on SpecTRM-GSCs, which aids in the development of autonomous systems. System performance can be tested through simulation before any hardware is built or any code is written. During maintenance, changes to the software can be easily documented and incorporated into the new system. Engineers can also simulate design alternatives for trade-off analyses as well as visualize the underlying state machine to obtain a different perspective of the system. Most important, SpecTRM-GSCs grant users the benefits of reuse without the potential drawbacks. The next chapter provides an example of Component-Based Systems Engineering using SpecTRM as applied to a real spacecraft.

Chapter 4
SPHERES Example

SPHERES stands for Synchronize Position Hold Engage Reorient Experimental Satellites. It was created by the MIT's Space Systems Laboratory to provide NASA and the Air Force with a reusable, space-based test-bed for high-risk metrology, control and autonomy technologies [17]. These technologies are critical to the operation of distributed satellite and docking missions such as the Terrestrial Planet Finder and Orbital Express. In addition, guest scientists from around the world will have access to this test-bed so they can independently design, code and debug estimation, control and autonomy algorithms for testing in the micro-gravity conditions of space (SPHERES will operate aboard the International Space Station) [9]. The experiments performed by NASA, the Air Force and the guest scientists are an important step in making many future space missions possible, especially those that require the ability to autonomously coordinate and synchronize multiple spacecraft in tightly controlled spatial configurations.

SPHERES was chosen as the case study for testing Component-Based Systems Engineering for many important reasons. First, SPHERES is an autonomous system. It was created to execute maneuvers without the guidance of a ground controller. In fact, the only interactions the astronauts aboard the ISS have with the SPHERES system involves simply loading programs and replenishables onto the spheres. Since Component-Based Systems Engineering was developed to support autonomous spacecraft, it was important to test the technique on an autonomous system. Along the same lines, it was also important that the system be highly modular to test the component-based aspects of the technique. Third, and most essential to the research, was the need to test the technique on a real system. In order to evaluate the scalability and applicability of this method, it was critical to test the process on a real spacecraft system. These three criteria made SPHERES an excellent experimental subject for testing Component-Based Systems Engineering. This chapter describes the process of reverse-engineering SPHERES using the Component-Based Systems Engineering approach, which involved (1) outlining the structure of SPHERES, (2) creating SpecTRM-GSCs from the

SPHERES project, (3) creating two example Guest Scientist Programs and (4) performing analyses on the SpecTRM-GSCs.

4.1 SpecTRM-GSC Structure

As outlined in Chapter 2, the first step in performing Component-Based Systems Engineering is a functional decomposition of the spacecraft. Figure 3 illustrates the SPHERES decomposition. SPHERES can operate with one, two or three spheres.

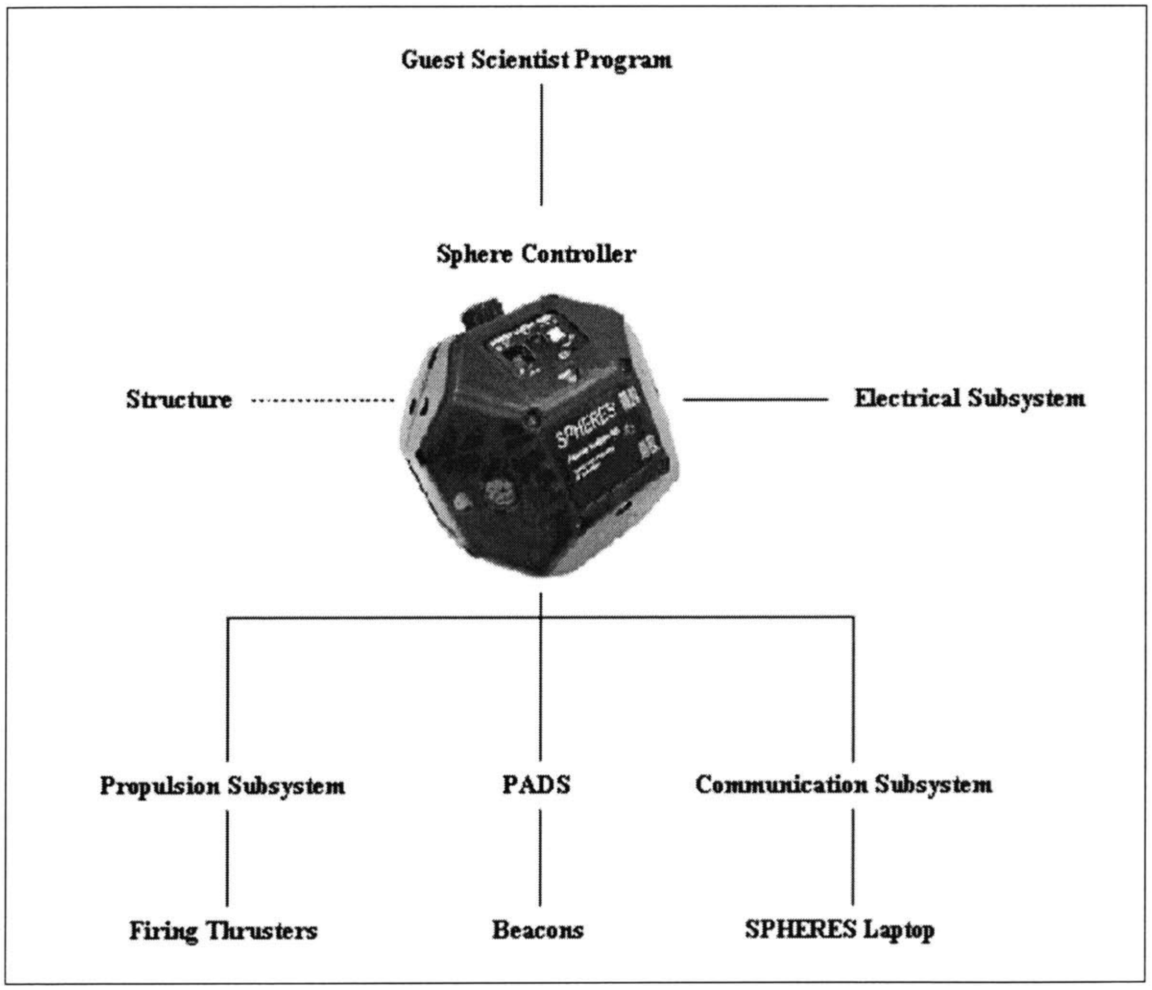


Figure 3. SPHERES Functional Decomposition

Each node in the decomposition tree represents a different SpecTRM-GSC, or intent specification. At the highest level is the Sphere Controller. The Sphere Controller provides the overall framework for the Sphere, coordinating the actions of the onboard components as well as determining the operating mode of the Sphere. Each Sphere Controller interacts with the various subsystems onboard the Sphere including Propulsion, Position and Attitude Determination, Communication, Guest Scientist Program, Electrical and Structure. The focus of this case study is on the subsystems that contain both hardware and software. Therefore, the Structure and Electrical Subsystems were not modeled, although they could easily be represented as SpecTRM-GSCs.

Each subsystem can then be broken down into its constituent hardware components. In the case of SPHERES, the majority of these components are too simple to model – they do not have sensors or provide any feedback to their parent subsystem nor are the hardware components redundant. Therefore, they are not shown in the decomposition diagram. The components that do contain their own software are listed in the diagram and include Firing Thrusters from the Propulsion Subsystem, Beacons from PADS and the SPHERES Laptop from the Communication Subsystem. Because of the simplicity of these models, intent specifications were not created for them in this case study. The SPHERES example focuses on the subsystem- and system-level intent specifications.

Each Sphere receives attitude and position information from its PADS, or Position and Attitude Determination Subsystem. PADS consists of three gyroscopes, three accelerometers, one ultrasound transmitter, five beacons and twelve sensing boards. Each sensing board contains two ultrasound receivers, one infrared receiver and two infrared transmitters. PADS receives angular acceleration from the three gyroscopes, linear acceleration from the three accelerometers and the Sphere Controller is able to calculate position and attitude information from the ranges between the fixed beacons and the Sphere's receivers. State estimation using the sensing boards and beacons is beyond the scope of this thesis. Therefore, the attitude information used in the example Guest Scientist Programs is provided entirely by the gyroscopes. Levels 1, 2 and 3 of the complete SpecTRM-GSC created for PADS can be found in Appendix B.

The Guest Scientist Program, or GSP, is written to perform either state estimation, control calculations or both to determine which control actions need to be performed to achieve a new position and/or attitude. There are two GSPs used in this test case: the Rate Damper and the Rate Matcher. These programs are described in further detail in Section 4.3.

The Propulsion Subsystem provides management of both position and attitude for the Sphere. The Propulsion Subsystem consists of twelve thrusters placed around the outside of each Sphere that simply turn on and off at calculated times. The Propulsion Subsystem is described in further detail in Section 4.2.

The individual Spheres can communicate with one another and a SPHERES Laptop through the Communication Subsystem, which consists of radio transmitters and receivers. There are two radios on each Sphere (providing two channels of communication) and one radio on the SPHERES Laptop. One radio on each Sphere provides Sphere-to-Sphere communication while the other provides Sphere-to-Laptop communication. Astronauts aboard the International Space Station load and download programs and information to and from the Spheres through this Laptop.

As previously stated, each node of the decomposition tree in Figure 3 represents a different intent specification. For this case study, intent specifications were created for each of the subsystems as well as for each Sphere Controller being used. The next section describes, in detail, an example of how a node of the decomposition was modeled as a SpecTRM-GSC.

4.2 Subsystem Example

This section provides a detailed description of the Propulsion Subsystem and how it was modeled as a SpecTRM-GSC. Each sphere relies on twelve on-off thrusters for position and attitude management. The geometry of the twelve thrusters on the sphere enables the production of force or torque using only two thrusters. The twelve thrusters are arranged in six pairs allowing for full six-degrees-of-freedom actuation. The propellant for the thrusters is compressed CO_2, which is fed through tubing from a high-pressure storage tank [9]. Appendix A provides complete Levels 1, 2 and 3 of the Propulsion Subsystem SpecTRM-GSC.

Level 0 of the specification was left blank in the generic sphere specifications, as it is particular to the organization and engineering team of the project and should therefore be written by the engineering team members. Level 1 of the specification includes the system-level goals, requirements and constraints. An example of a high-level functional requirement can be seen in Figure 4.

[FG.2] The Guest Scientist shall be able to turn thrusters on and off by either sending timed on/off commands to the Propulsion Subsystem or by sending the Propulsion Subsystem desired force and torque vectors. [→FR.1] [↓DP.1.2]
Rationale: This goal identifies the need to provide the Guest Scientist with an option to directly control the thrusters or to stock compute thruster firing times if he/she is not interested in performing the calculations in the Guest Scientist Program.

Figure 4. Level 1 System Goal

There are two links at the end of this requirement. In SpecTRM, these links are implemented as hyperlinks and can be easily created and changed. The first link points to the functional requirement, also at Level 1, which elaborates upon the system goal. Goals are too high level to be requirements. They are not necessarily testable, and there may be many system designs that meet the system's goals that are unacceptable due to other constraints. High-level requirements are generated from the goals and constraints present at this level. These are the "shall" statements that specify what the system is to do. These are the testable requirements.

The second link points down to a design principle at Level 2 that provides information about the design features that describe the different types of control the Guest Scientist can exert over the thrusters. It is also important to note that the rationale behind the requirements at every level is also recorded to ensure that future engineers working on the project understand why decisions were made instead of merely how the system works. In the example in Figure 4, the rationale for the system goal identifies the need to provide the Guest Scientist with options as to how the thrusters can be controlled. Scientists not interested in direct thruster control can merely send their desired forces and torques to the Propulsion Subsystem to have the thruster on and off times stock computed.

Because the components are created with a systems engineering approach, a safety analysis is also completed at each level of the system's development. At Level 1 a preliminary subsystem hazard analysis is completed. A hazard analysis at this level involves (1) defining an accident, (2) defining a safety policy, (3) creating a hazard list and classifying the hazards and (4) perform a hazard analysis using either fault trees, event trees or any other hazard analysis technique. For the SPHERES project, an accident is defined as any injury to one of the astronauts aboard the International Space Station or any damage to the SPHERES system that interferes with its ability to do science. Based on this accident definition, the following classifications can be applied to hazards to determine their associated severity:

- Level 1: Any injury to an astronaut or any damage to the SPHERES system that eliminates all ability to do science.
- Level 2: Damage to the SPHERES system that interferes with its ability to do science.
- Level 3: Damage to the SPHERES system that does not interfere with its ability to do science.

In addition, under NASA safety policy, each Sphere must be doubly fault-tolerant. This means that the Propulsion Subsystem must be able to withstand two faults [22]. Based on the accident definition and safety policy, the hazards seen in Figure 5 were identified for the Propulsion Subsystem [22].

[H.1] There is a pressure rise in the Propulsion Subsystem. [→SC.3] [↓DP.2.4]
Classification: Level 1
Rationale: A pressure rise in the Propulsion Subsystem may result in an explosion that may either injure an astronaut or damage the SPHERES system itself.

Figure 5. Level 1 Hazard Identification and Classification

Fault trees were used as the hazard analysis technique for the SPHERES project. The fault tree created for hazard [H.1] can be found in Appendix A Page 56. As seen in Figure 5, there are also links from the hazards down to the Level 4 Hardware Design Specifications that show how these hazards have been mitigated through hardware. Other Level 1 information consists of general background, historical information, environment descriptions, assumptions and constraints, system functional goals, operator requirements, interface requirements, design and safety constraints and information about the verification and validation requirements and results on the information at this level.

Level 2 of the intent specification specifies the design principles used to implement the Level 1 requirements. Figure 6 provides an example of the design principle linked to the functional requirement from Level 1 (Figure 4). It defines the different options the Guest Scientist has for controlling the thrusters in the Propulsion Subsystem. There are links to the corresponding functional goal and requirement. There is also a link to an element in the detailed logic specifications at Level 3 that models this design principle.

[DP.1.2] The Guest Scientist will decide which Control Mode the Propulsion Subsystem is in. If the Propulsion Subsystem is in Direct Mode, then the Guest Scientist provides the Propulsion Subsystem with timed on/off commands. If the Propulsion Subsystem is in Force Torque Mode, the Guest Scientist provides the Propulsion Subsystem with desired force and torque vectors. [↑FG.2] [↑FR.1] [↓PropulsionSubsystemControlMode]
Rationale: The Guest Scientist may want to directly control the thrusters or to stock compute thruster firing times if he/she is not interested in performing the calculations in the Guest Scientist Program.

Figure 6. Level 2 Design Principle

Another feature of the SpecTRM-GSCs allows users to indicate system-specific information that has to be changed given a component's particular instance of use. As seen in the design principle in Figure 7, the text "200ms" and "5Hz" is highlighted in boldface and underlined. This alerts the system engineer to change this portion of the requirement when this component is reused. In this instance, the Guest Scientist changes the sampling rate of the Propulsion Subsystem depending on the needs of his or her program. The sampling rate will affect how long each thruster must remain open to achieve the needed force and/or torque. Level 2 of the specification also includes system interface design, control and display design, operator task design principles and verification and validation requirements and results.

[DP.3.2.5] Once the thruster pair forces are determined, the thruster on and off times can be calculated. The thrust time is equal to the sampling rate of the Propulsion Subsystem divided by the ratio of the thrust provided by the thruster pair to the force needed from the thruster pair. The sampling rate of the Propulsion Subsystem is **200ms** and the force provided by the thruster pair is 0.2N. [↑EA.2]
*Rationale: The operating frequency is **5Hz**, which translates into a sampling rate of 200ms. Each thruster produces 0.1N of force and therefore the thruster pair produces 0.2N.*

Figure 7. Example of System Specific Information

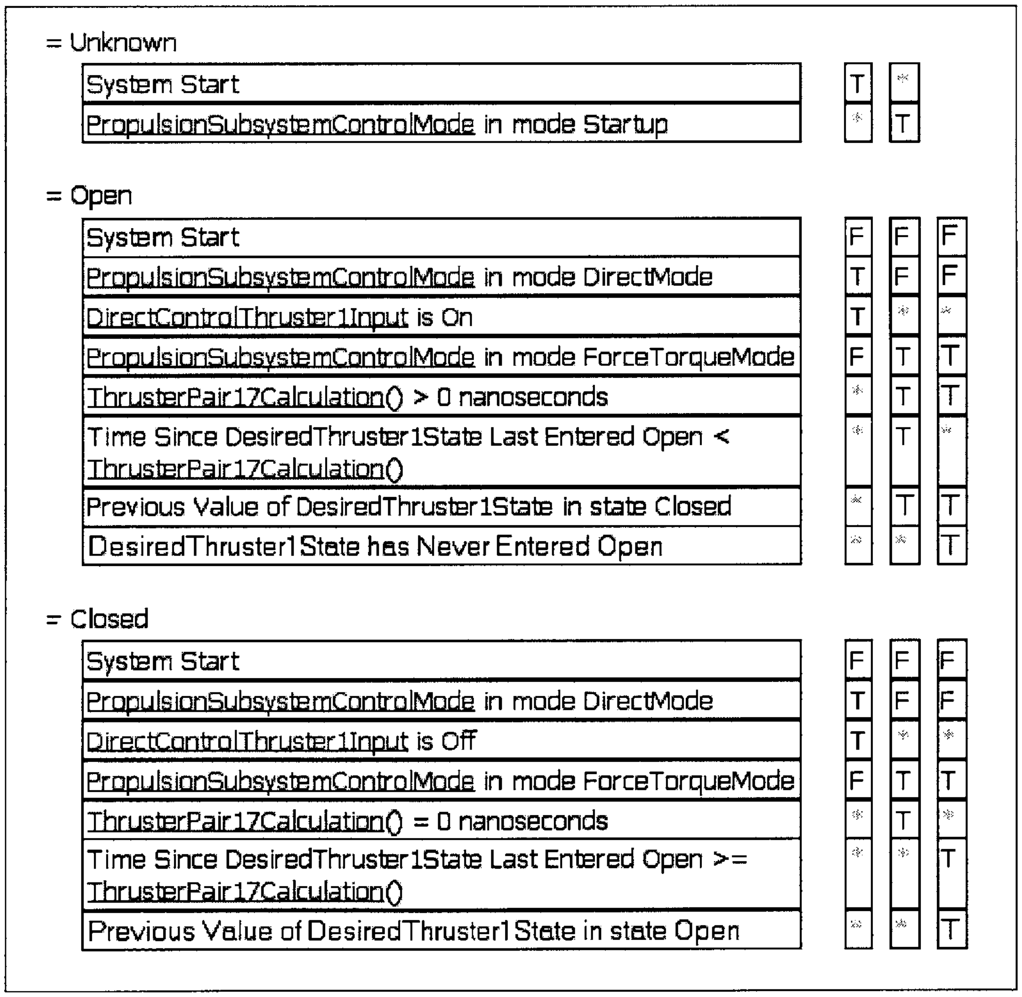
Level 3 of the intent specifications contains a formal, blackbox model of the component's externally visible behavior. The formal models, which are based on state machines, are specified using a language called SpecTRM-RL that was designed with reviewability and ease of learning as goals. Experience in using the language on industrial projects shows that engineers can learn to read SpecTRM-RL models with about ten to 15 minutes of training.

Appendix A Page 64 shows the graphical overview of the blackbox model of the Propulsion Subsystem. The graphical model depicts the control loops in which the Propulsion Subsystem is embedded. The left side shows the interface with the controller of the Propulsion Subsystem, which is the Sphere Controller. The right side of the diagram shows the hardware the Propulsion Subsystem is in turn controlling. The shaded part of the model describes the required behavior of the Propulsion Subsystem. There are three main parts of this description: the supervisory mode specifies the current controller of the component (in case there are multiple controllers); the current control mode for the component (Startup, ForceTorqueMode and DirectMode); and, to the right of the solid line, the controller's current state model of the controlled components. At any time, a controller only has a model, inferred from inputs and other information, about the real state of the components. In the Propulsion Subsystem example, the inferred state model has twelve state variables, representing information about the current desired state of the twelve thrusters. The graphical notation also shows the possible values for these state values; for example, the "SolenoidValve1State" state variable can have the values "Unknown," "Open" or "Closed." The boxes external to the gray-shaded area are devices external to the system that provide inputs to and take outputs from the blackbox. In this example, the Propulsion Subsystem interacts with the Sphere controller and the twelve thrusters.

The behavior of the Propulsion Subsystem, i.e. the logic for sending output commands to the various external devices and changing the inferred system state, is specified using a tabular notation called AND/OR tables. The rows of the tables indicate AND relationships, while the columns represent ORs. Figure 8 shows transition conditions required for the "SolenoidValve1State" state value to take the values "Unknown," "Open" and "Closed." Using the example AND/OR tables in Figure 8, the Propulsion Subsystem "SolenoidValve1State" element will transition to a new state if any of the columns in the transition table evaluate to true. In other words, if the "Thruster1" has received a direct on command from the Guest Scientist and the "PropulsionSubsystemControlMode" is in mode "DirectMode," then the "SolenoidValve1State" will transition to "Open." "Thruster 1" will also transition to "Open" if the "PropulsionSubsystemControlMode" is in mode "ForceTorqueMode" and the "ThrusterPair17Calculation" returns a duration greater than zero.

As seen in the three transition tables, there are several statements with an asterisk in the

OR column. This represents a "don't care" condition. In the example in Figure 8, the "SolenoidValve1State" state value will transition to "On" if the Propulsion Subsystem is in the DirectMode of control and the valve receives a direct on command. In this case, the valve "doesn't care" what the thruster pair calculations output. In other words, if the system is in the "DirectMode" of control we "don't care" about calculations only needed in the "ForceTorqueMode" in which the Propulsion Subsystem itself determines thruster on/off commands.



= Unknown

System Start	T	*
PropulsionSubsystemControlMode in mode Startup	*	T

= Open

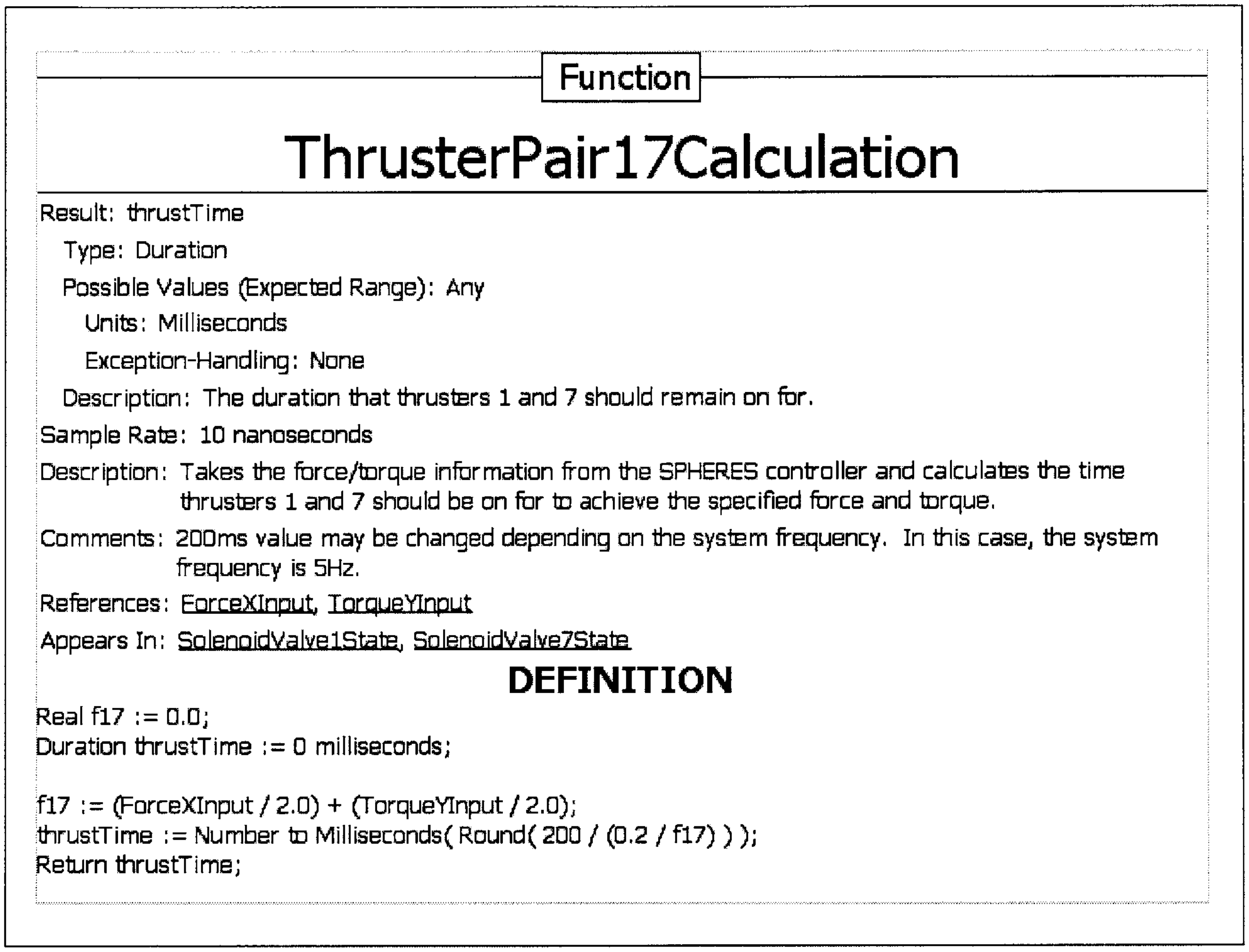
System Start	F	F	F
PropulsionSubsystemControlMode in mode DirectMode	T	F	F
DirectControlThruster1Input is On	T	*	*
PropulsionSubsystemControlMode in mode ForceTorqueMode	F	T	T
ThrusterPair17Calculation() > 0 nanoseconds	*	T	T
Time Since DesiredThruster1State Last Entered Open < ThrusterPair17Calculation()	*	T	*
Previous Value of DesiredThruster1State in state Closed	*	T	T
DesiredThruster1 State has Never Entered Open	*	*	T

= Closed

System Start	F	F	F
PropulsionSubsystemControlMode in mode DirectMode	T	F	F
DirectControlThruster1Input is Off	T	*	*
PropulsionSubsystemControlMode in mode ForceTorqueMode	F	T	T
ThrusterPair17Calculation() = 0 nanoseconds	*	T	*
Time Since DesiredThruster1State Last Entered Open >= ThrusterPair17Calculation()	*	*	T
Previous Value of DesiredThruster1 State in state Open	*	*	T

Figure 9. Level 3 And/Or Table

When the Propulsion Subsystem is in "ForceTorqueMode" it receives a force/torque vector from the Sphere Controller that specifies the needed forces and torques to accomplish a maneuver. The Propulsion Subsystem then calculates how long the thrusters need to be on to achieve the actuation. As defined in Figure 9, when the Propulsion Subsystem is in the "ForceTorqueMode" of control and the time since the valve was opened is greater than or equal to the calculated "Open" duration for the thruster, the valve will close. Figure 10 shows "ThrusterPair17Calculation" that determines this duration.


Function

ThrusterPair17Calculation

Result: thrustTime
  Type: Duration
  Possible Values (Expected Range): Any
    Units: Milliseconds
    Exception-Handling: None
  Description: The duration that thrusters 1 and 7 should remain on for.
Sample Rate: 10 nanoseconds
Description: Takes the force/torque information from the SPHERES controller and calculates the time thrusters 1 and 7 should be on for to achieve the specified force and torque.
Comments: 200ms value may be changed depending on the system frequency. In this case, the system frequency is 5Hz.
References: ForceXInput, TorqueYInput
Appears In: SolenoidValve1State, SolenoidValve7State

DEFINITION

```
Real f17 := 0.0;
Duration thrustTime := 0 milliseconds;

f17 := (ForceXInput / 2.0) + (TorqueYInput / 2.0);
thrustTime := Number to Milliseconds( Round( 200 / (0.2 / f17) ) );
Return thrustTime;
```

Figure 10. Level 3 Function Definition

Functions in SpecTRM-RL are written in an Ada-like programming script. This script allows users to calculate values from the inputs. In the case of the Propulsion Subsystem, functions are used to calculate the duration that valves should be "Open" to achieve the forces and torques needed by the Sphere Controller. Another SpecTRM element known as a macro allows users to abstract common logic and to increase readability. A macro takes a piece of an AND/OR table from another part of the model and gives it a name. This name can be substituted in for that table portion elsewhere in the model. The macro definition is a single AND/OR table. This table will evaluate to true or to false. When the table is true, the macro as a whole evaluates to true where it is used in other model elements. Similarly, if the table is false, then the macro evaluates to false.

The elements of the SpecTRM blackbox include output commands, output values, modes, states, macros, functions, command inputs and input values. Other information contained at Level 3 includes communication channels, operational procedures, user models and the results of performing various analyses and simulations on the blackbox model. Section 4.4 describes the simulations run on the SpecTRM-GSCs created for the SPHERES project.

4.3 Guest Scientist Program

As previously stated, the Guest Scientist Programs (GSPs) used in this test case are the Rate Damper and the Rate Matcher. The Rate Damper eliminates any angular rate applied to an individual sphere by applying a series of equations to the angular rates recorded by the gyroscopes. The GSP receives angular rates from the Sphere Controller, applies the equations to the angular rates in a function and then outputs force and torque vectors to the Sphere Controller.

Because these components were created with reusability in mind, they are highly generic. Therefore, different Guest Scientist Program blackbox models can be easily swapped in a new simulation. In addition, it becomes trivial to simulate multiple Spheres because the generic components already exist. Unlike the Rate Damper, the Rate Matcher utilizes two Spheres. The Leader in the Rate Matcher example measures its own angular rate and sends those measurements to the Follower Sphere through the Communication Subsystems. The Follower Sphere then matches its own angular rate to that of the Leader Sphere. The use of two Spheres in the Rate Matcher as opposed to just one Sphere in the Rate Damper illustrates the reuseability of the components and the ease with which components can be plugged together to simulate an entirely new spacecraft configuration. In this case, multiple Spheres interact in a more complicated Guest Scientist Program.

As stated in Levels 1 and 2, the Sphere Controller sends force and torque vectors or direct commands to the Propulsion Subsystem. If the Propulsion Subsystem is in "ForceTorqueMode" it uses the force and torque vectors to calculate thruster on and off times. If it is in

"DirectMode," the Propulsion Subsystem receives on and off times directly from the Sphere Controller and immediately sends on and off commands to the thrusters. In both example GSPs used in this test case, the Propulsion Subsystem operates in "ForceTorqueMode." The Propulsion Subsystem will calculate thruster on and off times based on forces and torques needed to either nullify or match the angular rate of a Sphere.

Levels 1, 2 and 3 of the Sphere Controller SpecTRM-GSC can be found in Appendix D. The blackbox model on Page 9 0 i llustrates t he S phere C ontroller's i nterface w ith t he g eneric components, including elements needed for the Rate Damper example. Each device in the system-level s pecification r epresents a nother i ntent specification. The box labeled Propulsion Subsystem represents the blackbox model that was created for the Propulsion Subsystem. During a system-level simulation, these models interact. For example, the outputs from the Sphere Controller to the Propulsion Subsystem device in the Sphere Controller model become the inputs in the Propulsion Subsystem model from the Sphere Controller device. The outputs from the Propulsion Subsystem to the Sphere Controller device in the Propulsion Subsystem model become the inputs from the Propulsion Subsystem device to the Sphere Controller in the Sphere Controller model. Blackbox models of the Propulsion Subsystem, PADS, Communication Subsystem and the two Guest Scientist Programs can be found in Appendices A, B, C and E respectively.

4.4 Simulation

Simulations in S pecTRM c an b e v isualized t hrough t he a nimation o f t he d iagram t hat represents the blackbox model of the system. As shown in the screen captures in Appendix F, the animation includes highlighting the current values of the state and mode elements in yellow and displaying the current values of inputs and outputs in blue text under the element names. Obsolete data values are shown in green text under the element names. Time is shown in the upper left hand corner of the visualization window. As the simulation time progresses, the input, output, state and mode values update to reflect the new data. The side bar lists all the element values of all the models in the simulation. The bar located on the bottom of the visualization provides an event log showing detailed timing information.

Data was collected for these simulations on the SPHERES frictionless test-table. Each Sphere used in the Rate Damper and Rate Matcher simulations were rotated on the test-table to generate angular velocity. The angular rates were sent to and recorded onto the SPHERES laptop. These measurements provide the angular rate input values to PADS models during the simulations.

The first Guest Scientist Program tested in the SpecTRM simulator was the Rate Damper. The Rate Damper zeroes any angular velocity experienced by the single Sphere. Appendix F Figure 1 shows a screen capture of the Rate Damper blackbox simulation. As described above, the figure shows the values of the input and output elements in blue text. The Rate Damper is running on the Sphere Controller and therefore "UserControl" is highlighted under Control Mode. Similarly, the current states of the Propulsion Subsystem and PADS ("ForceTorque" and "AccelerometerGyro" respectively) are also highlighted in yellow. The Sphere Controller receives the angular rate from PADS and transfers that information to the Rate Damper. The Rate Damper then determines the torques needed to zero the angular rate in a function. The resulting force and torque vectors are sent through the Sphere Controller to the Propulsion Subsystem.

Figure 2 in Appendix F shows the entire simulation environment. In the side bar the Propulsion Subsystem element list has been expanded to show the "Open" and "Close" commands that are sent to the individual thrusters. These "Open" and "Close" commands are determined by functions in the blackbox model. In the side bar of the screen capture, for example, it can be seen that the function "ThrusterPair17Calculation" returns "1 millisecond," which is implemented by the "DesiredSolenoidValuve1State" transitioning to "Open." As the thrusters open and close the angular rate decreases to zero, which is reflected in the successive angular rates from PADS during simulation.

Now that one Sphere has been successfully simulated, another Sphere is incorporated into a more complicated simulation to illustrate the reusability of the components. The second Guest Scientist Program is the Rate Matcher, which allows a Follower Sphere to match its angular rate with that of the Leader Sphere. Appendix F Figure 3 shows the Leader and Follower blackbox models during simulation. Figure 4 is a screen capture of the entire simulation environment during the Rate Matcher simulation. The visualizations are animated in the same manner as the previous example.

For the Rate Matcher example, the process of assembling another Sphere involved copying the subsystem models, pasting them into two folders and renaming each copy to begin with either "Leader" or "Follower." The "Leader" and "Follower" subsystem models were then assembled into a "LeaderSphereController" model and a "FollowerSphereController" model. These system-level models communicate through their respective Communication Subsystems. As seen in the simulation environment in Figure 4, the Leader Sphere sends its angular rates to the Follower Sphere. The Follower Sphere then uses the Leader Sphere's angular rates and its own angular rates to determine the forces and torques needed to match the angular rates of both Spheres. The angular rates measured by the Follower Sphere migrate toward the angular rates measured by the Leader Sphere as the simulation progresses.

4.5 Summary

The SPHERES system provides a case study for the application of Component-Based Systems Engineering. Because SPHERES is autonomous and highly modular, it was well suited for testing the scalability and applicability of this method. Intent specifications were written for each subsystem and the Sphere Controller. Rationale was captured at each level of their development and links provide traceability throughout the entire document. The subsystem models were created to be generic and therefore clearly indicate information that needs to be changed upon each subsystem's use in a particular instance. By modeling a Sphere's subsystems and controller as SpecTRM-GSCs, simulations could be run on the blackbox behavior of the entire system. In addition, simulating multiple Guest Scientist Programs and building multiple Sphere models shows the reusability of SpecTRM-GSCs.

Chapter 5
Conclusion

One of the major obstacles facing the aerospace industry today is the increasing complexity of spacecraft. This complexity is rooted in many new technologies and industry goals. First, software has become an integral part of spacecraft design, controlling not only the spacecraft hardware, but also the onboard science missions. Most engineers also believed that trading off the simplicity of hardware for the abilities of software would decrease this complexity. While software has become pervasive throughout most industries and seems to provide a solution to many logistical problems, it does not decrease the complexity of systems. In fact, the exploding number of states and transitions in software makes today's spacecraft far more complex and error-prone then their hardware-driven predecessors.

Second, the increasingly ambitious mission goals combined with the exploration of distant planets has uncovered the need for highly autonomous spacecraft control systems. The current motivation at NASA and its contractors to make spacecraft autonomous through the use of artificial intelligence merely compounds the complexity problem.

Third, the poor implementation of reuse and Component-Based Software Engineering has also increased the complexity of today's spacecraft. Many reusable software components contain extra functions so that they can be reused on many different projects. In addition, companies are often reluctant to share detailed documentation of these components for proprietary reasons. Consequently, the software components are improperly integrated into the rest of the project software. The complexity added by the poor implementation of reuse has led to spacecraft and mission losses. Although these technologies hold the promise of simplifying spacecraft development, their improper use has caused more damage than successes. One prime example of this misuse was the Mars Climate Orbiter loss discussed in Chapter 1.

In addition to the increasing complexity, the aerospace industry faces the many challenges inherent in spacecraft engineering. The loss of domain knowledge accompanying the retirement of the Apollo Era spacecraft engineers is of great concern to NASA and its contractors. Often rationale behind design decisions is not recorded or transferred to new personnel. These organizations need to provide a means through which knowledge can be transferred as individuals retire or move to other industries. The aerospace industry also faces the problems caused by miscommunication among multidisciplinary engineering teams. Mistakes are often made because engineers with different backgrounds do not share the same terminology, education or experience. There needs to be a common medium through which engineers from different disciplines can communicate clearly and effectively.

The aerospace industry is endeavoring to address the difficulties caused by added complexity and the challenging nature of spacecraft engineering itself in a culture of budget cuts and public disinterest. The solutions attempted in the 1990s, especially Faster, Better, Cheaper, did not provide spacecraft developers with the added productivity and success rate that was intended. Instead many of the spacecraft developed using this strategy were lost, costing NASA not only millions of dollars but also public trust and support. NASA is now faced with the question of how to develop the next generation of spacecraft under tight budget constraints without experiencing the drawbacks of the Faster, Better, Cheaper approach.

This thesis proposed a new method of spacecraft development known as Component-Based Systems Engineering that addresses the question. Component-Based Systems Engineering combines aspects of both systems engineering and Component-Based Software Engineering to reap the benefits of each technology without incurring some of the costs. Instead of reusing code, engineers reuse the development of requirements specifications, both informal and formal. In this approach, the reuse takes place before any detailed design is completed, any hardware built or any software coded.

Components are created in a systems engineering development environment known as SpecTRM. SpecTRM is a toolkit that allows users to create intent specifications, which assist engineers in managing the requirements, design and evolution process. The intent specification suggests a structure and a set of practices that makes the information contained in the document more easily used to support the project lifecycle. Intent specifications help engineers to (1) find specification errors early in product development so that they can be fixed with the lowest cost and impact to the system design, (2) increase the traceability of the requirements and design rationale throughout the project lifecycle and (3) incorporate required system properties into the design from the beginning rather than the end of development, which can be difficult and costly.

Component-Based Systems Engineering begins with a decomposition of the spacecraft, followed by a construction of components, subsystems and finally the entire system. These individual components are called SpecTRM-GSCs, or Generic Spacecraft Components, and provide users with a variety of benefits. The components are generic, which makes them highly reusable. Engineers can change project specific information based on the instance of the component's use. They also contain component-level fault protection, laying the foundation for a fault protection scheme that parallels the spacecraft's development. Finally, the formal portion of the SpecTRM-GSC can be analyzed individually or as part of their parent subsystem- and system-level before any implementation has taken place.

This thesis provided an example of Component-Based Systems Engineering as applied to the SPHERES system. SPHERES was chosen as the test case for evaluating Component-Based Systems Engineering because it consists of a series of autonomous spacecraft, forms a testbed for evaluating multiple different control and estimation algorithms, and is a real spacecraft system that will operate aboard the International Space Station. SpecTRM-GSCs were created for the various subsystems, including two Guest Scientist Programs, the Rate Damper and the Rate Matcher. Because the Rate Matcher used two Spheres while the Rate Damper used only one, simulations of both Guest Scientist Programs clearly illustrated the reusability of the SpecTRM-GSCs and the ease with which they can be combined to create another Sphere.

Through performing Component-Based Systems Engineering on SPHERES, it was shown how many of the problems outlined in Chapter 1 can be solved. The use of SpecTRM helps to solve the problems of domain knowledge capture through the recording of rationale at every stage of development. In addition, the use of SpecTRM-RL at Level 3 of the intent specification provides a readable and unambiguous formal specification that provides a common language with which engineers can easily communicate their requirements specifications.

SpecTRM-GSCs provide spacecraft engineers with a library of generic components that can be reused from one project to the next. Because the detailed design and implementation is has not been completed, engineering teams can tailor the components to fit their needs instead of fitting their needs to a particular piece of code. The use of component-level fault protection also encourages engineering teams to incorporate fault protection software into their designs from the beginning of the development process.

One area in which Component-Based Systems Engineering will be particularly beneficial is autonomous spacecraft. Instead of attempting to take the technological leap from time-stamped command sequences directly to artificial intelligence, Component-Based Systems

Engineering encourages the development of autonomous spacecraft through rigorous requirements specification development. These structured informal and formal specifications can be thoroughly analyzed for major inconsistencies and incompleteness before any implementation has occurred. In addition, special flight phases and faults can be easily simulated and the results of these simulations analyzed for deficiencies in the requirements specifications. Consequently, many errors and inadvertent omissions can be found early in the project lifecycle when they are less costly to correct. The fault-protection scheme also aids in the development of highly autonomous systems, because the consequences of possible faults are addressed throughout the project lifecycle.

The research on and the test case application of Component-Based Systems Engineering show its potential for use in developing the next generation of spacecraft. The benefits of using the technique span not only the engineering issues faced by today's spacecraft development teams but also the difficulties inherent in the aerospace industry. The results of this thesis merit further investigation into the use of Component-Based Systems Engineering for autonomous spacecraft.

References

[1] Bronsard, Francois, et al. "Toward Software Plug-and-Play." *Proceedings of the Symposium on Software Reusability.* Boston, MA. May 1997.

[2] Caldwell, Douglas W. and Savio N. Chau. *Spacecraft Information Systems: Principles and Practice.* Version 1.0. Online. 24 Jan 2002.

[3] Dulac, Nicolas, Thomas Viguier, Nancy Leveson, and Margaret-Anne Storey. "On the Use of Visualization in Formal Requirements Specification." *Proceedings of the International Conference on Requirements Engineering.* Essen, Germany. Sep 2002.

[4] Enright, John. Personal Conversation. July 2003.

[5] Euler, Edward A., Steven D. Jolly, and H.H. 'Lad' Curtis. "The Failures of the Mars Climate Orbiter and Mars Polar Lander: A Perspective from the People Involved." *Advances in the Aeronautical Sciences, Guidance and Control.* Volume 107, 2001.

[6] Gat, Erann. "The MDS Autonomous Control Architecture." Jet Propulsion Laboratory. Pasadena, CA.

[7] Heimdahl, Mats P.E. and Nancy Leveson. "Completeness and Consistency Analysis of State-Based Requirements." *IEEE Transactions on Software Engineering.* May 1996.

[8] Hilstad, Mark O. "A Multi-Vehicle Testbed and Interface Framework for the Development and Verification of Separated Spacecraft Control Algorithms" Master's Thesis. Massachusetts Institute of Technology, Cambridge, MA. June 2002.

[9] Hilstad, Mark O., John P. Enright and Arthur G. Richards. "The SPHERES Guest Scientist Program Interface." Version 1.0. 20 Nov 2002.

[10] Honeywell. "Domain-Specific Software Architectures for GN&C." Online. 25 Jan 2002.

[11] International Council on Systems Engineering. "What is Systems Engineering?" Online. 2 June 2003.

[12] Keesee, John Col. USAF and Col. Peter Young USAF. Personal Conversation. July 2002.

[13] Leveson, Nancy. "Completeness in Formal Specification Language Design for Process-Control Systems." *Proceedings of Formal Methods in Software Practice Conference.* Portland, OR. Aug 2000.

[14] Leveson, Nancy. "Intent Specifications: An Approach to Building Human-Centered Specifications." *IEEE Transactions on Software Engineering.* Jan 2000.

[15] Lions, J.L., et al. *Ariane 5 Flight 501 Failure: Report by the Inquiry Board.* Paris, France. 19 July 1996.

[16] Massachusetts Institute of Technology. The ESD Symposium Committee. *ESD Terms and Definitions*. Version 12. 19 Oct 2001.

[17] Massachusetts Institute of Technology. Space Systems Lab. "SPHERES Science CDR." 18 Nov 2002.

[18] National Aeronautics and Space Administration. NASA Chief Engineer and NASA Integrated Action Team. *Enhancing Mission Success – A Framework for the Future.* 21 Dec 2001.

[19] National Aeronautics and Space Administration. *NASA Systems Engineering Handbook.* June 1995.

[20] National Aeronautics and Space Administration. Office of Inspector General. *Audit of Faster, Better, Cheaper: Policy, Strategic Planning, and Human Resource Alignment.* 13 Mar 2001.

[21] NASA/ESA Investigation Board. *SOHO Mission Interruption.* 31 Aug 1998.

[22] Nolet, Simon. "SPHERES Propulsion System."

[23] Pfahler, Peter, Uwe Kastens. "Configuring Component-Based Specifications for Domain-Specific Languages." *Proceedings of the 34th Hawaii International Conference on System Sciences*. 2001.

[24] Rasmussen, Robert D. "Goal-Based Fault Tolerance for Space Systems Using the Mission Data System." *IEEE*. 2001.

[25] Safeware Engineering Corporation. *SpecTRM User Manual.* Version 1.0.0. 2003.

[26] Van Vliet, Hans. *Software Engineering Principles and Practice.* Second Ed. New York: Wiley, 2001.

[27] Weyuker, Elaine. "Testing Component-Based Software: A Cautionary Tale." *IEEE Software.* September/October 1998.

Appendix A

Propulsion Subsystem

Level 1: System-Level Goals, Requirements, and Constraints

Introduction

The Propulsion Subsystem aboard each Sphere in the SPHERES system (Synchronize Position Hold Engage Reorient Experimental Satellites) provides management of both position and attitude for the Sphere. The Propulsion Subsystem consists of a series of thrusters that can be turned on and off depending on commands received from the Sphere's controller.

Historical Information

This intent specification is a SpecTRM-GSC (Generic Spacecraft Component). It should only be used for the SPHERES project and encompasses only information relating to the Propulsion Subsystem.

Environment Description

There will be only two types of devices in the Propulsion Subsystem's environment with which it must communicate:

Sphere Controller - The Sphere Controller is the system-level intent specification in the SpecTRM-GSC decomposition of SPHERES. It provides commands to the Propulsion Subsystem.

Firing Circuits - There are twelve firing circuits in the Propulsion Subsystem hardware (one for each solenoid valve). These are component-level intent specifications and they command the actual solenoid valves to open and close.

Environment Assumptions

[EA.1] This Propulsion Subsystem is operating within a Sphere.
[↓System Interface Design]
Rationale: This intent specification describes a propulsion subsystem that will operate within a Sphere. It is not meant for use outside the SPHERES project.

[EA.2] The software used in the Propulsion Subsystem is operating at a frequency of 1kHz. [↓DP.3.2.5]
Rationale: The Propulsion Subsystem software must be able to accommodate different control frequencies and implement pulse modulation.

System Functional Goals

[FG.1] The Propulsion Subsystem shall provide forces and torques for the Sphere. [↓DP.1] [↓DP.2] [↓DP.3]
Rationale: Each Sphere requires a subsystem that will actuate the maneuvers that are determined by the Guest Scientist Program.

[FG.2] The Guest Scientist shall be able to turn thrusters on and off by either sending timed on/off commands to the Propulsion Subsystem or by sending the Propulsion Subsystem desired force and torque vectors. [→FR.1] [↓DP.1.2]
Rationale: This goal identifies the need to provide the Guest Scientist with an option to directly control the thrusters or to stock compute thruster firing times if he/she is not interested in performing the calculations in the Guest Scientist Program.

High-Level Requirements

[FR.1] The Propulsion Subsystem shall receive either timed on/off commands or desired force and torque vectors from the Sphere Controller. [→FG.1] [↓DP.1.2]
Rationale: This goal identifies the need to provide the Guest Scientist with an option to directly control the thrusters or to stock compute thruster firing times if he/she is not interested in performing the calculations in the Guest Scientist Program.

[FR.2] The Propulsion Subsystem shall send an "On" command to the firing circuits at the requested time when it receives a direct "On Time" command from the Sphere Controller. [↓DP.3.1]
Rationale: A direct command from the Guest Scientist will be sent to the firing circuits that actuate the solenoid valves.

[FR.3] The Propulsion Subsystem shall send an "Off" command to the firing circuits at the requested time when it receives a direct "Off Time" command from the Sphere Controller. [↓DP.3.1]
Rationale: A direct command from the Guest Scientist will be sent to the firing circuits that actuate the solenoid valves.

[FR.4] If force and torque vectors are received from the Sphere Controller, the Propulsion Subsystem shall determine on and off times for each thruster based on the thrusters location and the force and torques needed. [↓DP.3.2]
Rationale: The Propulsion Subsystem shall provide the Guest Scientist with the ability to stock compute firing times for the thrusters based on desired force and torque vectors.

[FR.5] The Propulsion Subsystem shall send an "On" command to the firing circuits at the requested time if the calculations performed on the force and torque vectors determine that a thruster is needed at a specific time. [↓DP.3.1]
Rationale: The calculations performed by the Propulsion Subsystem on the force and torque vectors will generate "On Time" commands that will be sent to the firing circuits that actuate the solenoid valves.

[FR.6] The Propulsion Subsystem shall send an "Off" command to the firing circuits at the requested time if the calculations performed on the force and torque vectors determine that a thruster is needed at a specific time. [↓DP.3.1]
Rationale: The calculations performed by the Propulsion Subsystem on the force and torque vector will generate "Off Time" commands that will be sent to the firing circuits that actuate the solenoid valves.

[FR.7] The Propulsion Subsystem shall have enough thrusters to provide actuation throughout the six-degrees-of-freedom. [↓DP.1.3]
Rationale: The SPHERES system operates in space and therefore must be able to translate in three dimensions and rotate in three directions.

Design and Safety Constraints

Non-Safety Constraints

[SC.1] The Propulsion Subsystem must operate independently of any operator action. [↓DP.3]
Rationale: The SPHERES system is autonomous and therefore must operate without human interference.

[SC.2] The propellant tank used in the Propulsion Subsystem must not be empty. [→OR.2] [↓OT.1]
Rationale: The Propulsion Subsystem requires propellant to perform actuation.

Safety Constraints

[SC.3] There must be a mechanical system within each Sphere that will mitigate a pressure rise in the Propulsion Subsystem. [↓DP.2.4]
Rationale: [→H.1]

Operator Requirements

[OR.1] The operator shall monitor an estimate of the level of fuel in the tank. [↓CD.1]
Rationale: The propellant tanks contain a limited amount of fuel.

[OR.2] The SHPERES operator shall be able to replace the propellant tank when the amount of propellant drops below a pre-specified percentage. [→SC.2] [↓OT.1]
Rationale: Some maneuvers may not be able to be completed with a nearly depleted propellant tank.

System Interface Requirements

[IR.1] There shall be a means for an operator to estimate the level of fuel in the propellant tank. [↓CD.1]
Rationale: When the level of fuel in the propellant tank drops below a specified amount the tank needs to be replaced.

System Limitations

[L.1] The thrusters in the Propulsion Subsystem will not provide thrust if the tank is empty.

[L.2] There are no sensors that determine the current state of each thruster.
Rationale: The SPHERES system is too small to have sensors that monitor the thrusters.

[L.3] There are opening and closing transients associated with the solenoid valves.

Hazard List and Hazard Log

Accident Definition

An accident is defined as any injury to one of the astronauts on the International Space Station or any damage to any part of the SPHERES system that interferes with its ability to do science. From this accident definition, the following accident classification is used to determine the severity of the system hazards:

Level 1: Any injury to an astronaut or damage to the SPHERES system that completely eliminates all ability to do science.
Level 2: Damage to the SPHERES system that interferes with its ability to do science.
Level 3: Damage to the SPHERES system that does not interfere with its ability to do science.

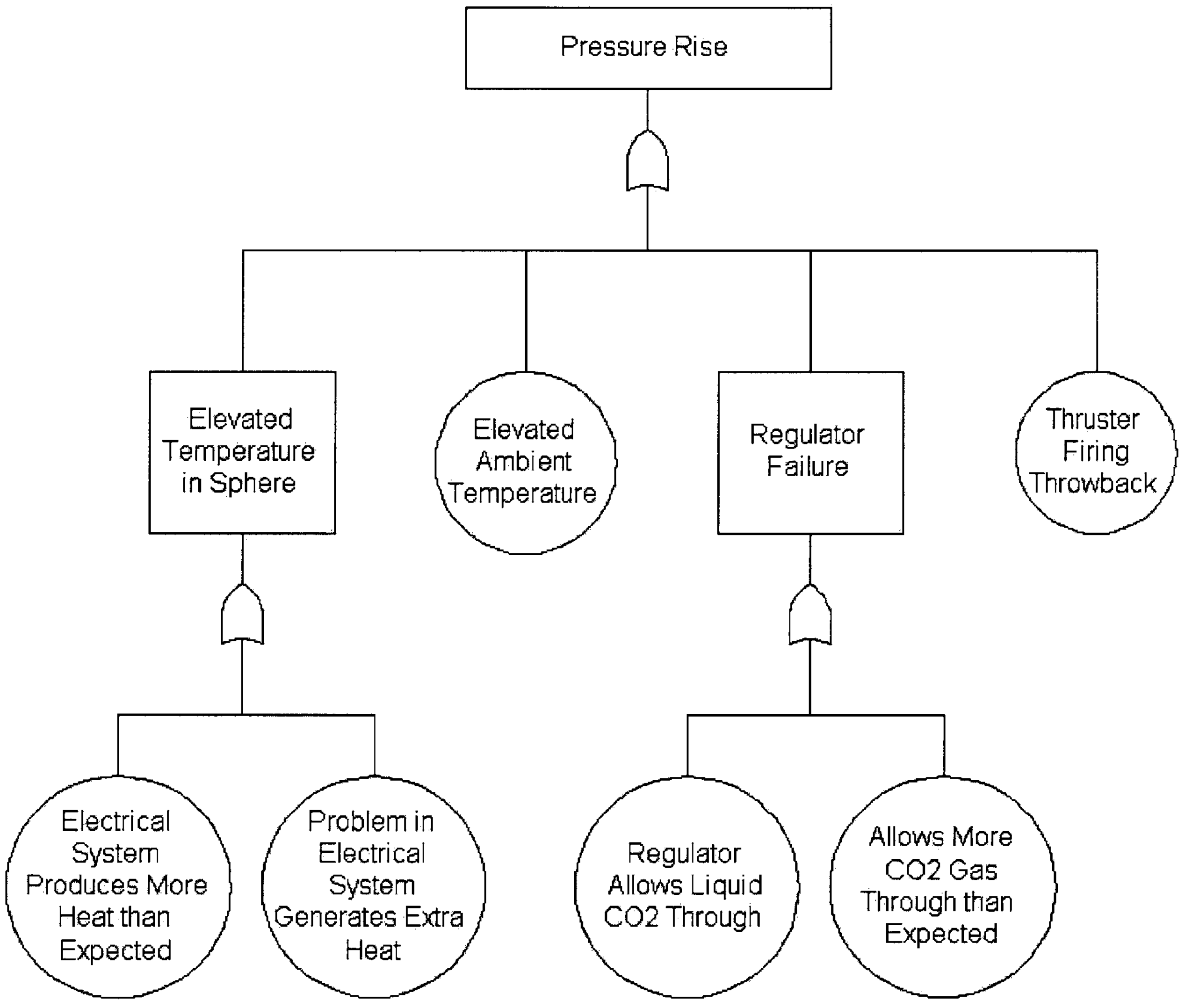
Safety Policy

Under NASA requirements, the SPHERES system must be doubly fault-tolerant. This means that at any point in the Propulsion Subsystem, the system must be able to sustain two faults.

Hazard List and Assessment

[H.1] There is a pressure rise in the Propulsion Subsystem. [→SC.3] [↓DP.2.4]
Classification: Level 1
Rationale: A pressure rise in the Propulsion Subsystem may result in an explosion that may either injure an astronaut or damage the SPHERES system itself.

Hazard Analysis



Pressure Rise
Elevated Temperature in Sphere
Elevated Ambient Temperature
Regulator Failure
Thruster Firing Throwback
Electrical System Produces More Heat than Expected
Problem in Electrical System Generates Extra Heat
Regulator Allows Liquid CO2 Through
Allows More CO2 Gas Through than Expected

Verification and Validation

Review Procedures

A review board that is independent of the development team verifies levels 1, 2 and 3 of the SPHERES system SpecTRM-GSCs. Multiple iterations of this review process can be performed to help ensure that the analysis reflects the actual operation of the

SPHERES system. Suggestions for improvement are added as necessary and evaluated in the next iteration of the review process.

Participants

Kathryn Weiss – SpecTRM-GSC Developer
John Enright – SPHERES Team Member and Primary Reviewer

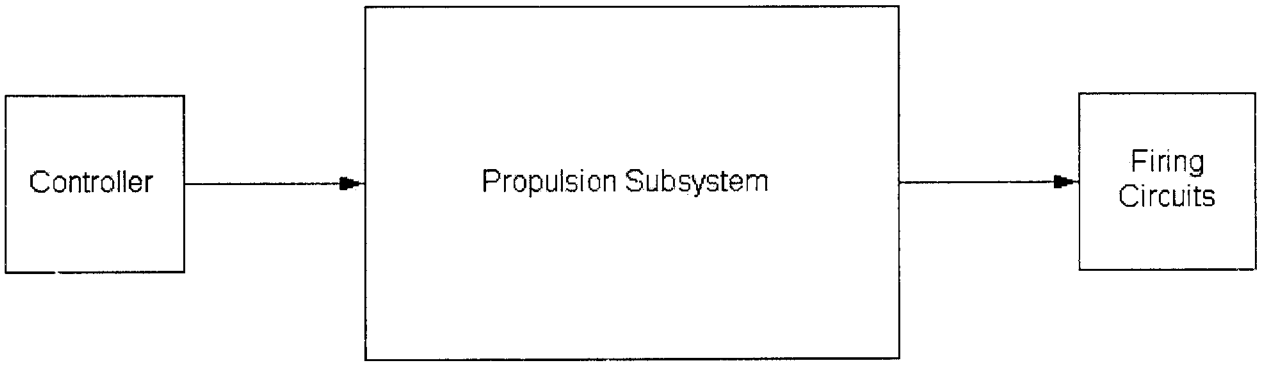
Results

This SpecTRM-GSC is the result of two development and review cycles.

Level 2: System Design Principles

System Interface Design

The external components will interface with the Propulsion Subsystem in the following manner: [↑EA.1]



The Propulsion Subsystem interfaces with the Sphere Controller and with the firing circuits that provide actuation to the twelve solenoid valves. Although the firing circuits are technically part of the Propulsion Subsystem, they are separated here because there is a SpecTRM-GSC for the firing circuits. [↓C.1]

The Sphere Controller sends the Propulsion Subsystem either timed on/off commands for each of the twelve thrusters or force and torque vectors. The Propulsion Subsystem does not send any information to the Sphere Controller. [↓C.2] [↓C.3]

The Propulsion Subsystem sends on and off commands to the firing circuits. The firing circuits do not send any information to the Propulsion Subsystem. [↓C.4]

Controls and Displays

Displays

[CD.1] The percentage of fuel left in the propellant tank is displayed on the SPHERES Laptop. Please see the SPHERES Laptop specification for details on how this is implemented. [↑IR.1] [↑OR.1]
Rationale: The SPHERES Laptop is the interface through which the operator interacts with the SPHERES system.

Operator Task Design Principles

[OT.1] The operator shall replace the propellant tank that is currently in the Sphere if the tank is less than 20% full. This amount roughly translates to changing the propellant tank 3-4 times per hour. [↑OR.1] [↑SC.2] [↓OP.1]
Rationale: Operators should not run out of fuel in the middle of a test.

System Design Principles

[DP.1] Propulsion Subsystem Overview [↑FG.1]

[DP.1.1] The Propulsion Subsystem is made up of software and hardware components. The software components consist of thruster control and pulse modulation. The hardware components consist of the firing circuits, solenoid valves, regulator and capacitor, tubing and manifolds, nozzles and propellant tank. Figure 1 provides a diagram of the Propulsion Subsystem.

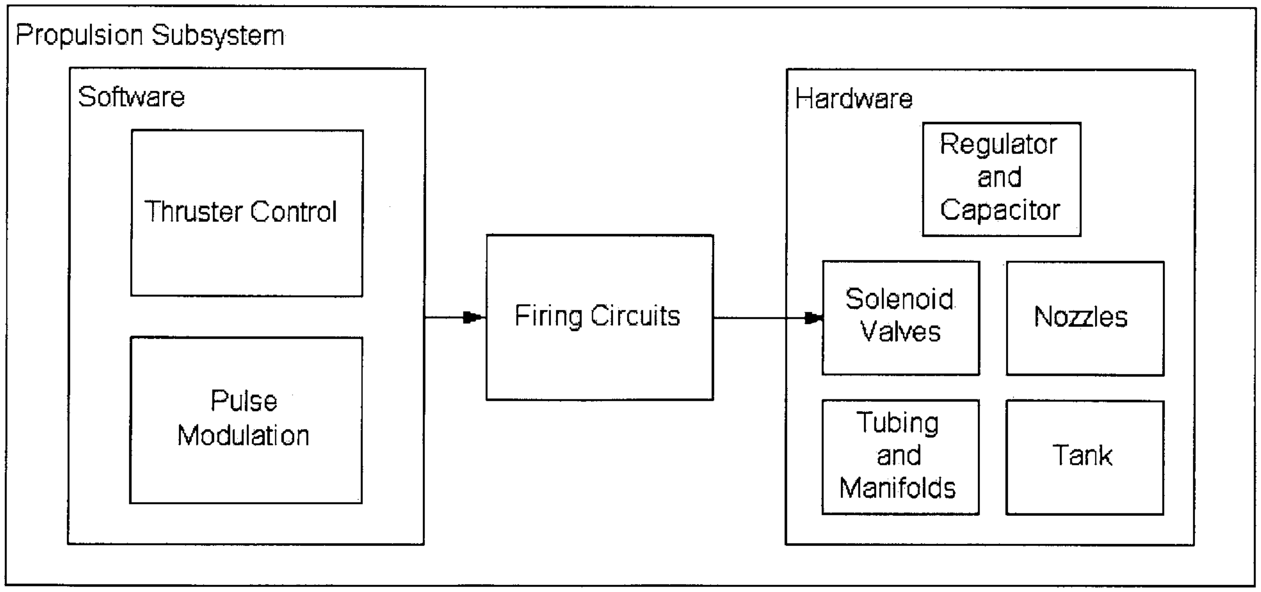


Figure 1. Propulsion Subsystem Diagram

[DP.1.2] The Guest Scientist controls the Control Mode of the Propulsion Subsystem. If the Propulsion Subsystem is in Direct Mode, then the Guest Scientist provides the Propulsion Subsystem with timed on/off commands. If the

Propulsion Subsystem is in Force Torque Mode, the Guest Scientist provides the Propulsion Subsystem with desired force and torque vectors. [↑FG.2] [↑FR.1] [↓PropulsionSubsystemControlMode]
Rationale: The Guest Scientist may want to directly control the thrusters or to stock compute thruster firing times if he/she is not interested in performing the calculations in the Guest Scientist Program.

[DP.1.3] The thrusters are arranged on the Sphere in order to provide pure body-axis force or torque using only two thrusters, assuming uniform mass and inertia properties. The twelve thrusters are arranged in six back-to-back pairs, allowing for full six-degrees-of-freedom actuation. Figure 2 shows the Sphere thruster configuration. Figure 3 shows the directions of the force and torque that will be produced by firing each thruster based on the configuration in Figure 2. The combinations of thrusters required to produce force along or torque about each body axis are shown in Figure 4. [↑FR.7]
Rationale: It was expected that the majority of maneuvers would involve primarily body-axis rotations, and the flight thruster geometry is significantly more propellant-efficient than other geometries for these maneuvers.

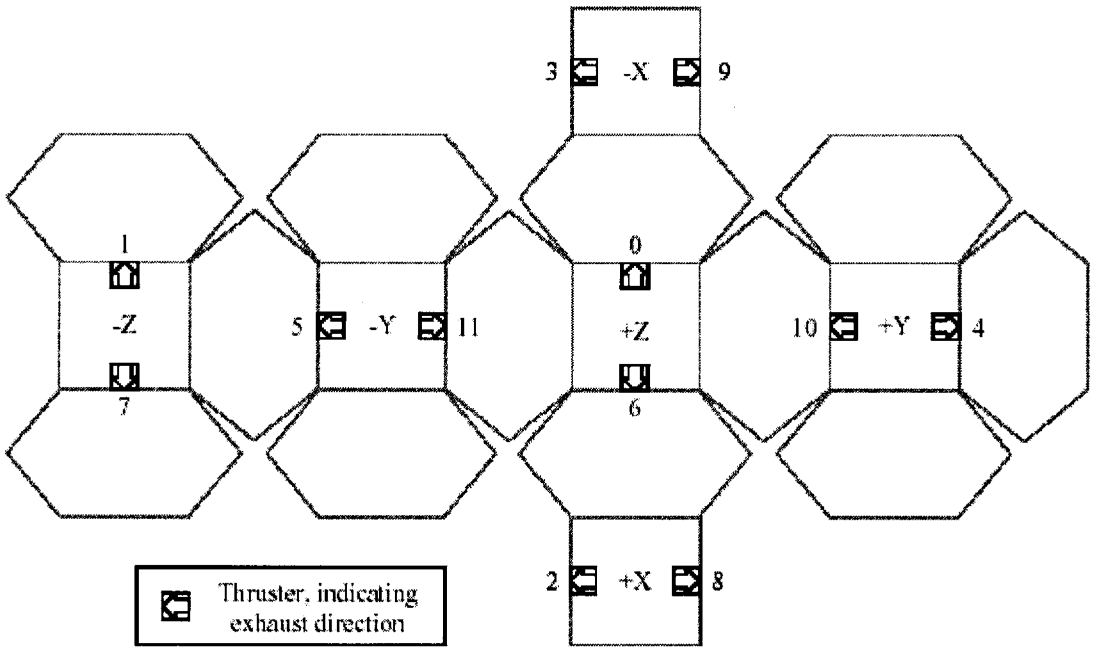


Figure 2. Thruster Pair Configuration

Thr #	Thruster position [cm]			Resultant force direction			Resultant torque direction		
	x	y	z	x	y	z	x	y	z
0	-5.16	0.0	9.65	1	0	0	0	1	0
1	-5.16	0.0	-9.65	1	0	0	0	-1	0
2	9.65	-5.16	0.0	0	1	0	0	0	1
3	-9.65	-5.16	0.0	0	1	0	0	0	-1
4	0.0	9.65	-5.16	0	0	1	1	0	0
5	0.0	-9.65	-5.16	0	0	1	-1	0	0
6	5.16	0.0	9.65	-1	0	0	0	-1	0
7	5.16	0.0	-9.65	-1	0	0	0	1	0
8	9.65	5.16	0.0	0	-1	0	0	0	-1
9	-9.65	5.16	0.0	0	-1	0	0	0	1
10	0.0	9.65	5.16	0	0	-1	-1	0	0
11	0.0	-9.65	5.16	0	0	-1	1	0	0

Figure 3. Thruster Force and Torque Directions

Thr #	Body-axis force						Body-axis torque					
	+x	-x	+y	-y	+z	-z	+x	-x	+y	-y	+z	-z
1	×								×			
2	×									×		
3			×								×	
4			×									×
5					×		×					
6					×			×				
7		×								×		
8		×							×			
9				×								×
10				×							×	
11						×		×				
12						×	×					

Figure 4. Thruster Combinations

[DP.2] Propulsion Subsystem Hardware [↑FG.1]

[DP.2.1] There are twelve firing circuits – one for each of the twelve thrusters. The firing circuits generate an opening waveform and current amplification from a digital command.

[DP.2.2] There are twelve solenoid valves - one for each of the twelve thrusters. The solenoid valves open in response to a firing command from the firing circuits. [↓DesiredThruster1State] [↓DesiredThruster2State] [↓DesiredThruster3State] [↓DesiredThruster4State] [↓DesiredThruster5State] [↓DesiredThruster6State] [↓DesiredThruster7State] [↓DesiredThruster8State]

[↓DesiredThruster9State] [↓DesiredThruster10State]
[↓DesiredThruster11State] [↓DesiredThruster12State]

[DP.2.2.1] The solenoid valves remain open as long as the firing circuits continue to send a firing command.

[DP.2.3] Each propellant tank contains up to 172g of pressurized CO2 stored in liquid form at 860 psi.
Rationale: 172g is the maximum amount of CO2 that can be stored in the propellant tank that fits inside of each Sphere at 860 psi.

[DP.2.4] There are two pressure release mechanisms, or burst disks in the Propulsion Subsystem. One is attached to the tank coupling and one is on the regulator itself. These mechanisms burst if the pressure builds to greater than 4500 psi. [↑SC.3] [↑H.1]
Rationale: The burst disks will rupture before the tank reaches a hazardous pressure of greater than 4500 psi thereby releasing the building pressure.

[DP.2.5] The regulator is used to expand the liquid CO2 into a gas and simultaneously decrease the thruster feed pressure to between 0 and 35 psig.
Rationale: At 35 psig the average thruster force is approximately 0.1N, which is the desired operating thrust for each thruster.

[DP.2.6] The capacitor stores low-pressure gas and thereby helps to maintain a constant working pressure under different propellant flow rates (i.e. when different numbers of thrusters are open) throughout the Propulsion Subsystem.

[DP.2.7] The nozzles provide a choked sonic flow.
Rationale: This aerodynamic effect is used to maximize thrust.

[DP.2.8] The tubing and manifolds provide transportation of propellant, CO2, from the propellant tank to each of the twelve thrusters.

[DP.3] Propulsion Subsystem Software [↑FG.1] [↑SC.1]

[DP.3.1] Thruster Control [↑FR.2] [↑FR.3] [↑FR.5] [↑FR.6]

[DP.3.1.1] The Thruster Control software handles the timing of firing commands.

[DP.3.1.2] The Thruster Control software sends "On" and "Off" commands to the firing circuits. These commands can either come directly from the Sphere Controller "On Time" and "Off Time" commands or from the Pulse Modulation portion of the Propulsion Subsystem Software.

[DP.3.2] Pulse Modulation [↑FR.4] [↓ThrusterPair17Calculation] [↓ThrusterPair28Calculation] [↓ThrusterPair39Calculation] [↓ThrusterPair410Calculation] [↓ThrusterPair511Calculation] [↓ThrusterPair612Calculation]

[DP.3.2.1] The Pulse Modulation software calculates the duration that thrusters should be opened based on body-referenced force and torque vectors from the Sphere Controller.

[DP.3.2.2] The Pulse Modulation software receives x, y and z force and torque values from the SPHERES Controller.

[DP.3.2.3] The Propulsion Subsystem uses the following flight SPHERES thruster combinations to produce pure body-axis force or pure body-axis torque about a particular axis. For example, from Figure 4, net force in the x direction results from a combination of thrusters 1 and 7, where thruster 1 is in the positive x direction and thruster 7 is in the negative x direction.

f1,7 = f1 - f7
f2,8 = f2 - f8
f3,9 = f3 - f9
f4,10 = f4 - f10
f5,11 = f5 - f11
f6,12 = f6 - f12

Rationale: Mark Hilstad's Master's Thesis

[DP.3.2.4] The Propulsion Subsystem uses the following equations that map the force and torque commands to thruster pair forces in the flight Sphere geometry. The moment arm r is equal to 9.7 cm.

$$\begin{bmatrix} f_{1,7} \\ f_{2,8} \\ f_{3,9} \\ f_{4,10} \\ f_{5,11} \\ f_{6,12} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & 0 & 0 & -\frac{1}{2} & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 & 0 & 0 & -\frac{1}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 \end{bmatrix} \begin{bmatrix} f_x \\ f_y \\ f_z \\ t_x \\ t_y \\ t_z \end{bmatrix}$$

Rationale: Mark Hilstad's Master's Thesis

[DP.3.2.5] Once the thruster pair forces are determined, the thruster on and off times can be calculated. The thrust time is equal to the sampling rate of the Propulsion Subsystem divided by the ratio of the thrust provided by the thruster pair to the force needed from the thruster pair.

The sampling rate of the Propulsion Subsystem is **200ms** and the force provided by the thruster pair is 0.2N. [↑EA.2]
Rationale: The operating frequency is ***5Hz****, which translates into a sampling rate of 200ms. Each thruster produces 0.1N of force and therefore the thruster pair produces 0.2N.*

Level 3: Blackbox Behavior

Communication

[C.1] The Sphere Controller sends the Propulsion Subsystem the control mode it should enter, either Force Torque Mode or Direct Mode. [↑System Interface Design] [→GuestScientistModeInput]

[C.2] When the Propulsion Subsystem is in Force Torque Mode, the Sphere Controller sends it Forces (x, y, z) and Torques (x, y, z). [↑System Interface Design] [→ForceXInput] [→ForceYInput] [→ForceZInput] [→TorqueXInput] [→TorqueYInput] [→TorqueZInput]

[C.3] When the Propulsion Subsystem is in Direct Mode, the Sphere Controller sends it On Time or Off Time commands for each of the twelve thrusters. [↑System Interface Design] [→DirectControlThruster1Input] [→DirectControlThruster2Input] [→DirectControlThruster3Input] [→DirectControlThruster4Input] [→DirectControlThruster5Input] [→DirectControlThruster6Input] [→DirectControlThruster7Input] [→DirectControlThruster8Input] [→DirectControlThruster9Input] [→DirectControlThruster10Input] [→DirectControlThruster11Input] [→DirectControlThruster12Input]

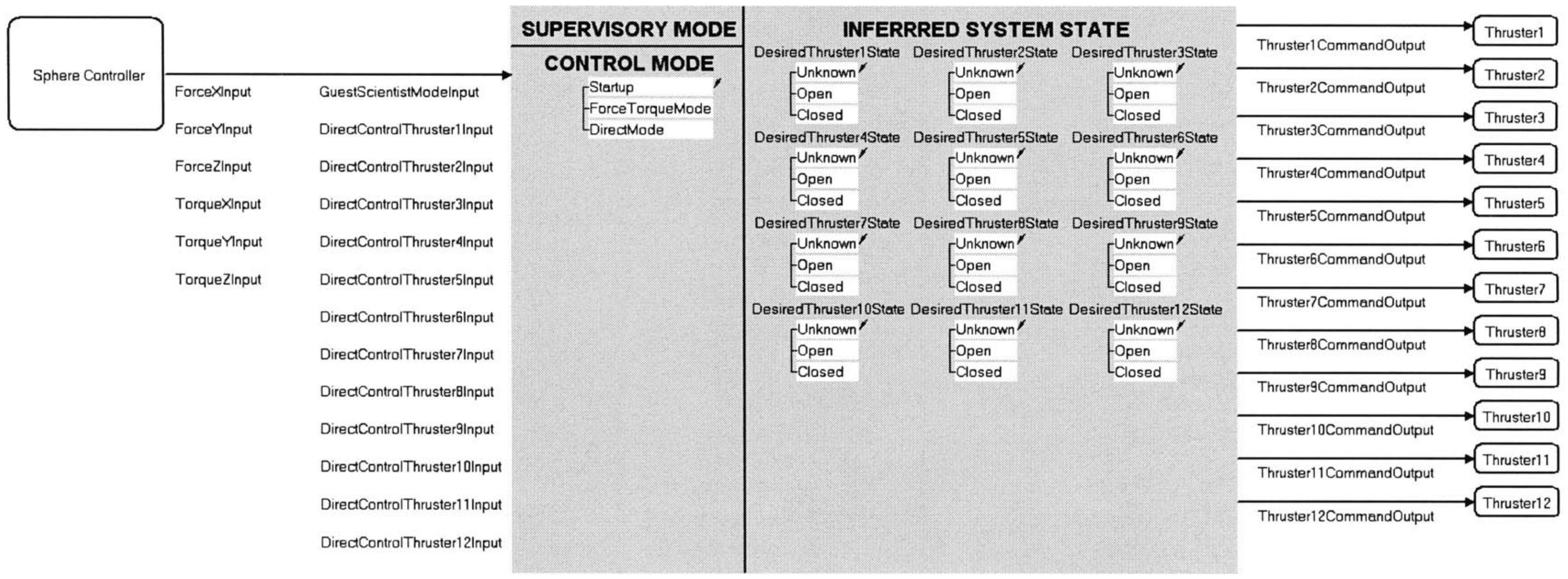
[C.4] The Propulsion Subsystem sends the Firing Circuits On or Off commands. [↑System Interface Design] [↓Thruster1CommandOutput] [↓Thruster2CommandOutput] [↓Thruster3CommandOutput] [↓Thruster4CommandOutput] [↓Thruster5CommandOutput] [↓Thruster6CommandOutput] [↓Thruster7CommandOutput] [↓Thruster8CommandOutput] [↓Thruster9CommandOutput] [↓Thruster10CommandOutput] [↓Thruster11CommandOutput] [↓Thruster12CommandOutput]

Operational Procedures

[OP.1] Replacing a propellant tank. [↑OT.1]

[OP.1.1] Fully open regulator.
[OP.1.2] Vent tank.
[OP.1.3] Change tank.
[OP.1.4] Mark old tank.
[OP.1.5] Reset CO2 level.
[OP.1.6] Resume testing.

System Blackbox Behavior - Propulsion Subsystem



Appendix B

PADS

Level 1: System-Level Goals, Requirements, and Constraints

Introduction

The Position and Attitude Determination Subsystem (PADS) aboard each Sphere in the SPHERES system (Synchronize Position Hold Engage Reorient Experimental Satellites) provides state estimation for the Sphere. PADS provides direct measurements of linear acceleration and angular rate. PADS also provides time-of-flight measurements to fixed beacons to calculate position and attitude.

Historical Information

This intent specification is a SpecTRM-GSC (Generic Spacecraft Component). It should only be used for the SPHERES project and encompasses only information relating to the Position and Attitude Determination Subsystem.

Environment Description

There will be three types of devices in the PADS environment with which it must communicate:

Sphere Controller - The Sphere Controller is the system-level intent specification in the SpecTRM-GSC decomposition of SPHERES. It provides commands to PADS and receives position and attitude information from PADS.

Inertial Measurement System - This system consists of accelerometers and gyroscopes that provide the Sphere with measurements of linear acceleration and angular velocity sampled at a user-specified rate. These are component-level devices that do not have intent specifications because they are pure hardware.

Global Measurement System - Provides the Sphere with a measurement of time of flight to each fixed beacon in the SPHERES operating space. The Guest Scientist can compute any position and attitude information from these ranges. The beacons are represented by component-level SpecTRM-GSCs.

Environment Assumptions

[EA.1] PADS is operating within a Sphere. [↓System Interface Design]
Rationale: This intent specification describes a position and attitude determination subsystem that will operate within a Sphere. It is not meant for use outside the SPHERES project.

Environment Constraints

[EC.1] PADS must operate within an enclosed 1.5m x 1.5m x 2m volume. [↓DP.3.2.3]
Rationale: This is the size of the enclosed volume of the United States Node on the International Space Station. A known volume is needed so that a beacon chirp used in the Global portion of PADS disperses before the next beacon chirps.

[EC.2] This ultrasound beacons are placed at known locations around the inside of Node 1. [→IR.1] [→OR.1]
Rationale: Accurate position and attitude information is contingent on receiving accurate readings from the beacons surrounding the SPHERES operating space.

System Functional Goals

[FG.1] PADS shall provide accurate calculation of Sphere position and/or attitude information. [↓DP.1] [↓DP.2] [↓DP.3]
Rationale: Accurate position and/or attitude information is needed so that the Guest Scientist Program or Sphere Controller can determine the required actuation.

[FG.2] The Guest Scientist can choose whether to use an inertial measurement system, a global measurement system or both systems to receive needed state information. [→FR.1] [↓DP.1.2]
Rationale: The inertial measurement system provides measurements much faster than the global measurement system. The two systems provide different rates for processing the information.

High-Level Requirements

[FR.1] PADS receives a message from the Sphere Controller that indicates which type of position and attitude information is needed. [→FG.2] [↓DP.1.2]
Rationale: Different Guest Scientist Programs require different position and attitude information.

[FR.2] PADS shall provide the Sphere Controller with linear acceleration and angular velocity with the inertial measurement system. [↓DP.2.1] [↓DP.3.1]
Rationale: The Guest Scientist may not want to use the global measurement system.

[FR.3] PADS shall provide the Sphere Controller with ranges from the Sphere to each beacon in the SPHERES operating space. [↓DP.2.2] [↓DP.3.2]
Rationale: Any position and/or attitude information needed by the Guest Scientist can be calculated from these ranges. In addition, the Guest Scientist may not want to use the inertial measurement system.

Design and Safety Constraints

Non-Safety Constraints

[SC.1] PADS must operate independently of any operator action. [↓DP.3]
Rationale: The SPHERES system is autonomous and therefore must operate without human interference.

Operator Requirements

[OR.1] The Operator shall place the beacons around the inside of Node 1 in known locations and record these locations. [→EC.2] [↓OT.1]
Rationale: The beacons are needed for the global measurement system. Because the beacons are not fixed inside of Node 1, the operator is responsible for placing the beacons in known locations before any SPHERES operation.

System Interface Requirements

[IR.1] There shall be a means for an operator to record the position of the beacons inside Node 1. [→EC.2] [↓OT.2]
Rationale: The beacons are needed for the global measurement system. Because the beacons are not fixed inside of Node 1, the operator is responsible for placing the beacons in known locations before any SPHERES operation.

System Limitations

[L.1] Accurate position information will not be reported unless the beacons are placed at known locations around the inside of Node 1 and the locations are recorded.
Rationale: The beacons are needed for the global measurement system.

[L.2] The elements of the inertial measurement system are not redundant.
Rationale: The SPHERES system is too small to have redundant components.

[L.3] The global measurement system is susceptible to interference from external sources of infrared and ultrasound. Presence of this interference will degrade performance.

Verification and Validation

Review Procedures

A review board that is independent of the development team verifies levels 1, 2 and 3 of the SPHERES system SpecTRM-GSCs. Multiple iterations of this review process can be performed to help ensure that the analysis reflects the actual operation of the SPHERES system. Suggestions for improvement are added as necessary and evaluated in the next iteration of the review process.

Participants

Kathryn Weiss – SpecTRM-GSC Developer
John Enright – SPHERES Team Member and Primary Reviewer

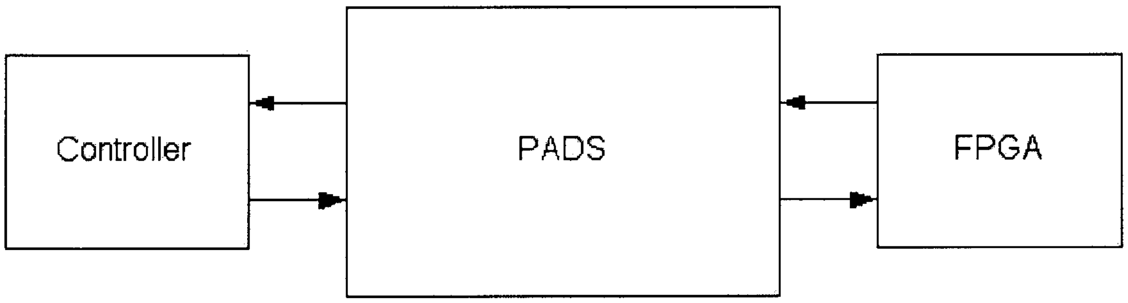
Results

This SpecTRM-GSC is the result of two development and review cycles.

Level 2: System Design Principles

System Interface Design

The external components will interface with the PADS in the following manner: [↑EA.1]



PADS interfaces with the Sphere Controller and with the FPGA, which provides an interface with the avionics hardware. Although the FPGA is technically part of the PADS, it is separated here because the inputs to and outputs from PADS are processed by the FPGA. There is no SpecTRM-GSC model of the FPGA because it was created by Payload Systems, Inc. and there is insufficient information to create an intent specification.

The Sphere Controller tells PADS which information is needed from the avionics hardware. The Sphere Controller can either receive linear acceleration and/or angular velocity from the inertial measurement system and/or ranges to each beacon from the global measurement system. PADS sends the requested information to the Sphere Controller. [↓C.1] [↓C.2] [↓C.3] [↓C.4]

PADS sends a flash command to the FPGA, which begins the process of global metrology. PADS receives linear acceleration and angular velocity from the FPGA. [↓C.5] [↓C.6] [↓C.7] [↓C.8] [↓C.9] [↓C.10]

Controls and Displays

Displays

[OD.1] The SPHERES Laptop contains a display that allows the operator to enter the locations of the five ultrasound beacons. [↑IR.1]
Rationale: The Laptop is used during the experiment setup so that the beacon locations are loaded onto the Spheres, which use these locations in position determination.

Operator Task Design Principles

[OT.1] The Operator places the beacons at known locations, as in Figure 1, around the inside of Node 1. The beacons are placed on opposite alternate corners and the remaining beacon in any remaining corner. [↑OR.1] [↓OP.1]
Rationale: The Sphere requires the locations of the beacons in order to obtain accurate position information. The beacons are placed so that maximum coverage is obtained.

[OT.2] The Operator enters the location of the beacons into the SPHERES Laptop. [↑IR.1] [↓OP.2]
Rationale: The Sphere requires location of the beacons in order to obtain accurate position information.

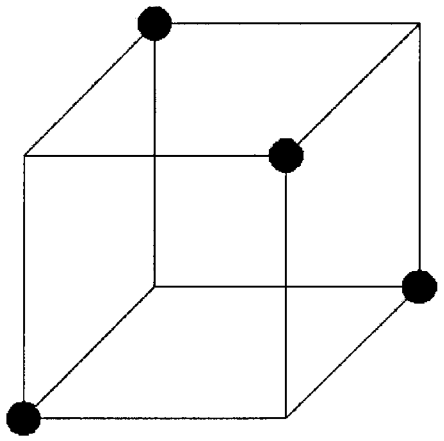


Figure 1. Example Beacon Configuration for the First Four Beacons

System Design Principles

[DP.1] Position and Attitude Determination Subsystem (PADS) Overview [↑FG.1]

[DP.1.1] PADS is made up of software and hardware components. The hardware components consist of the FPGA, gyroscopes, accelerometers, infrared transmitters and receivers, ultrasound transmitters and receivers and beacons. Figure 2 provides a diagram of PADS.

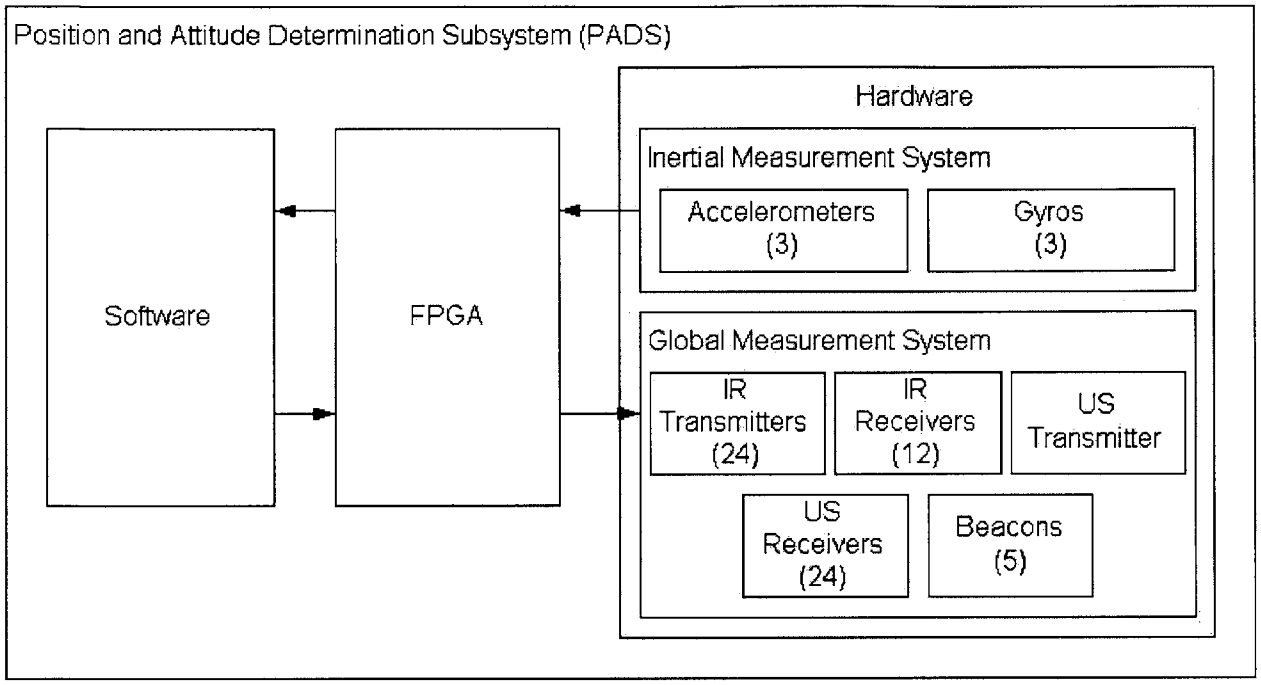


Figure 2. PADS Diagram

[DP.1.2] The Guest Scientist will decide which Control Mode PADS is in. If PADS is in Report Ranges Mode, PADS will issue a command to the FPGA to flash the infrared transmitter, beginning the process of global metrology. Ranges to each beacon are then sent to the Sphere Controller. If PADS is in Accelerometer and Gyro Mode it will send the measurements taken at a pre-specified sample rate by the FPGA. If PADS is in Send All Data mode, it will send information from both the global and inertial measurement systems. [↑FG.2] [↓PADSControlMode]
Rationale: Different Guest Scientists have different position and attitude information needs. For example, some Guest Scientists Programs involve position information, requiring them to use the global measurement system, while others need quickly sampled angular rates, in which case they can use the inertial measurement system.

[DP.2] PADS Hardware [↑FG.1]

[DP.2.1] Inertial Measurement System [↑FR.2]

[DP.2.1.1] The gyroscopes are mounted in alignment with the body axes at the positions listed in Figure 3.
Rationale: These locations provide easy and intuitive mapping of gyroscope measurements to Sphere motion.

[DP.2.1.2] The accelerometers are aligned parallel to, but displaced from, the body axes at the positions listed in Figure 4.

Rationale: Ideally, the accelerometers would be mounted along the three axes of the Sphere body frame, but this arrangement is not feasible given the spatial requirements of other subsystems.

Sensor	Location (body frame) [cm]		
	x	y	z
x-axis gyro	TBD	3.10	6.39
y-axis gyro	-5.49	TBD	-3.24
z-axis gyro	-5.49	3.24	TBD

Figure 3. Gyroscope Locations

Sensor	Location (body frame) [cm]		
	x	y	z
x-axis accelerometer	5.19	2.17	3.27
y-axis accelerometer	-2.66	3.35	3.30
z-axis accelerometer	3.28	-4.37	3.35

Figure 4. Accelerometer Locations

[DP.2.1.3] The FPGA continuously samples the gyroscopes and accelerometers at 1kHz. The FPGA converts the analog signal from the hardware components to a digital signal. The FPGA then sends this digital signal to PADS. If the Guest Scientist uses the accelerometer and gyro data, he/she down-samples the information at a specified rate.
Rationale: The Guest Scientist sets how often the data needs to be sampled.

[DP.2.2] Global Measurement System [↑FR.3]

[DP.2.2.1] The Global Measurement System hardware is comprised of 24 ultrasound receivers, 1 ultrasound transmitter, 12 infrared receivers, 24 infrared transmitters and 5 beacons. The ultrasound receivers, infrared receivers and infrared transmitters are split into 12 boards. Each board contains 2 ultrasound receivers, 1 infrared receiver and 2 infrared transmitters. Figure 5 illustrates the locations of the boards on the Sphere. Figure 6 provides the numbering scheme for the ultrasound receivers as well as their exact location in the body frame.

[DP.2.2.2] The five fixed beacons provide position and attitude relative to the walls of the operational module. Each fixed beacon contains an ultrasound transmitter and an infrared receiver.
Rationale: The infrared receiver begins the clock on the beacon after the Sphere flashes and the ultrasound transmitter sends out a chirp and the pre-specified time.

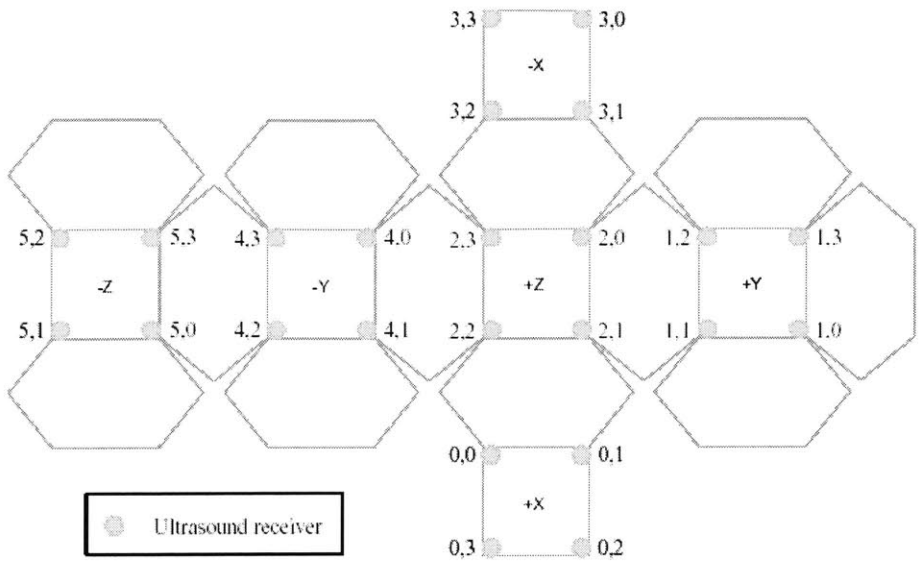


Figure 5. Ultrasound Receiver Geometry and Numbering

Face		Receiver	Location (body frame) [cm]		
Label	Number	Number	x	y	z
+x	0	0	10.23	-3.92	3.94
	0	1	10.23	3.92	3.94
	0	2	10.23	3.92	-3.94
	0	3	10.23	-3.92	-3.94
+y	1	0	3.94	10.23	-3.92
	1	1	3.94	10.23	3.92
	1	2	-3.94	10.23	3.92
	1	3	-3.94	10.23	-3.92
+z	2	0	-3.92	3.94	10.26
	2	1	3.92	3.94	10.26
	2	2	3.92	-3.94	10.26
	2	3	-3.92	-3.94	10.26
-x	3	0	-10.23	3.92	-3.94
	3	1	-10.23	3.92	3.94
	3	2	-10.23	-3.92	3.94
	3	3	-10.23	-3.92	-3.94
-y	4	0	-3.94	-10.23	3.92
	4	1	3.94	-10.23	3.92
	4	2	3.94	-10.23	-3.92
	4	3	-3.94	-10.23	-3.92
-z	5	0	3.92	-3.94	-10.23
	5	1	3.92	3.94	-10.23
	5	2	-3.92	3.94	-10.23
	5	3	-3.92	-3.94	-10.23

Figure 6. Ultrasound Sensor Numbering Scheme and Locations

[DP.3] PADS Software [↑FG.1] [↑SC.1]

[DP.3.1] Inertial Measurement System [↑FR.2]

[DP.3.1.1] PADS converts the digital signal (converted by the FPGA from the analog signal taken from the gyroscopes and accelerometers) into angular velocity in radians per second and linear acceleration in meters per second squared.

[DP.3.1.2] If PADS is operating in Accelerometer and Gyro Mode or in Send All Data Mode, then PADS will report these values to the Sphere Controller.

[DP.3.2] Global Measurement System [↑FR.3]

[DP.3.2.1] The Guest Scientist begins the PADS global metrology process by commanding PADS into Report Ranges Mode. PADS commands the infrared transmitters to flash. The infrared receivers sense the flash and notify PADS, which starts a clock.

[DP.3.2.2] PADS sends a message to the Sphere Controller to turn off all thrusters.
Rationale: Thrusters produce ultrasound noise that will cause false triggering of global measurement system events.

[DP.3.2.3] 10ms after the infrared is received, Beacon 1 sends out an ultrasonic signal, or a chirp. 30ms after the infrared is received, Beacon 2 chirps. 50ms after the infrared is received, Beacon 3 chirps. 70ms after the infrared is received, Beacon 4 chirps. 90ms after the infrared is received, Beacon 5 chirps. [↑EC.1]
Rationale: Given the size of US Node 1 and the positions of the beacons around the inside of that volume, these times ensure that the beacon chirps will not overlap.

[DP.3.2.4] The ultrasound receivers send a "received" message to PADS when they receive a chirp. PADS calculates a time-of-flight between the time the beacon chirped and when the receivers sensed the chirp. The time since the ultrasound receiver senses the beacon transmission minus the time since the beacon chirps is the time of flight. PADS calculates the range between each beacon and each ultrasound receiver. Range is equal to the time-of-flight divided by the speed of sound, which is **340m/s** (meters per second).
Rationale: 340m/s is the speed of sound at room temperature. The SPHERES Laptop allows the Operator to record the actual ambient temperature so that the speed of sound can be more accurately calculated.

[DP.3.2.5] PADS finds time-of-flight to another Sphere using the onboard beacon (the ultrasound transmitter).
Rationale: Sphere-to-Sphere range and bearing are used for Guest Scientist Programs that involve docking.

Level 3: Blackbox Behavior

Communication

[C.1] The Sphere Controller sends PADS what Control Mode it should enter, either Report Ranges Mode, Accelerometer and Gyro Mode or Send All Data Mode. [↑System Interface Design] [→GuestScientistModeInput]

[C.2] When PADS is in Report Ranges Mode, it sends the Sphere Controller ranges from each of the 24 ultrasound receivers to each of the 5 beacons in meters. [↑System Interface Design]

[C.3] When PADS is in Accelerometer and Gyro Mode, it sends the Sphere Controller Angular Velocity (x, y, z) in radians per second and Linear Acceleration (x, y, z) in meters per second squared. [↑System Interface Design] [→LinearAccelerationOutput] [→AngularRateOutput]

[C.4] When PADS is in Send All Data Mode, it sends the Sphere Controller both ranges from each of the 24 ultrasound receivers to each of the 5 beacons in meters and Angular Velocity (x, y, z) in radians per second and Linear Acceleration (x, y, z) in meters per second squared. [↑System Interface Design] [→LinearAccelerationOutput] [→AngularRateOutput]

[C.5] PADS sends a Flash command to the Infrared Transmitters through the FPGA. [↑System Interface Design]

[C.6] PADS sends a Flash command to the Ultrasound Transmitter through the FPGA. [↑System Interface Design]

[C.7] The Accelerometers send Linear Acceleration (x, y, z) to PADS through an FPGA. [↑System Interface Design] [→AccelerometerXInput] [→AccelerometerYInput] [→AccelerometerZInput]

[C.8] The Gyroscopes send Angular Acceleration (x, y, z) to PADS through an FPGA. [↑System Interface Design] [→GyroXInput] [→GyroYInput] [→GyroZInput]

[C.9] The Infrared Receivers send Infrared Received to PADS. [↑System Interface Design]

[C.10] The Ultrasound Receivers send Time Beacon Flash Received to PADS.
[↑System Interface Design]

Operational Procedures

[OP.1] Placing the beacons. [↑OT.1]

[OP.1.1] Unstow five ultrasound beacons.

[OP.1.1] Place five beacons at the pre-specified seat-tracks as seen in Figure 6.

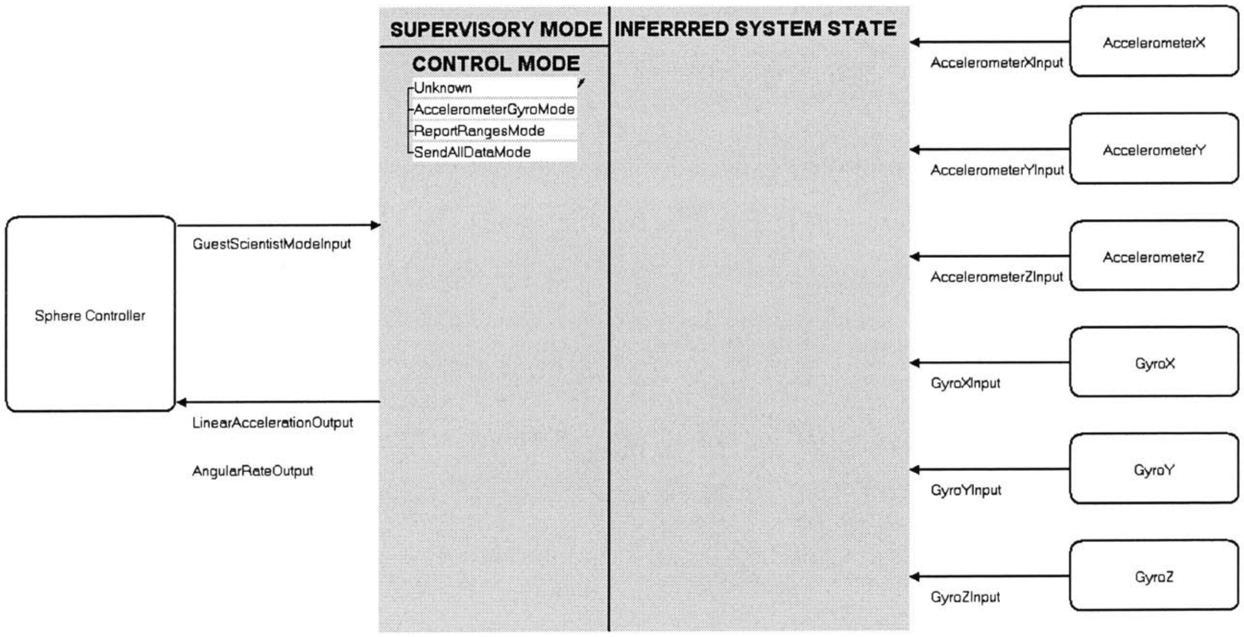
[OP.2] Record locations of five beacons in the SPHERES Laptop setup screen. [↑OT.2]

[OP.2.1] Enter beacon seat-track locations for each of the five beacons.

[OP.2.2] Save the beacons' positions.

[OP.3] Stow beacons after experiment has been completed.

System Blackbox Behavior – PADS



Appendix C

Communication Subsystem

Level 1: System-Level Goals, Requirements, and Constraints

Introduction

The Communication Subsystem aboard each Sphere in the SPHERES system (Synchronize Position Hold Engage Reorient Experimental Satellites) provides communications management. The Communication Subsystem provides wireless data transfer between the Spheres and between the Spheres and the ground. It is used to send user commands, coordinate actions of the Spheres and send telemetry.

Historical Information

This intent specification is a SpecTRM-GSC (Generic Spacecraft Component). It should only be used for the SPHERES project and encompasses only information relating to the Communication Subsystem.

Environment Description

There will be only four types of devices in the Communication Subsystem's environment with which it must communicate:

Sphere Controller - The Sphere Controller is the system-level intent specification in the SpecTRM-GSC decomposition of SPHERES. It provides commands and telemetry to the Communication Subsystem.

Other Spheres - There can be up to two other Spheres executing a Guest Scientist Program in the SPHERES system. These are other system-level intent specifications that the Communications Subsystem may need to communicate with.

SPHERES Laptop - Programs are loaded onto the Spheres through the SPHERES Laptop. The SPHERES Laptop commands the Spheres to run tests. In addition, telemetry is sent to the SPHERES Laptop per user specification. This is component-level intent specification.

Radios - All communications are transmitted and received through radios on board each Sphere. These are component-level devices and do not have intent specifications because they are pure hardware. The radios also provide different channels for communication.

Environment Assumptions

[EA.1] The Communication Subsystem is operating within a Sphere. [↓System Interface Design]
Rationale: This intent specification describes a communication subsystem that will operate within a Sphere. It is not meant for use outside the SPHERES project.

Environment Constraints

[EC.1] The Communication Subsystem must adhere to ISS EMI standards.
Rationale: The SPHERES system will operate aboard the International Space Station and therefore must adhere to their standards.

System Goals

[FG.1] The Communication Subsystem shall provide wireless data transfer abilities for the Sphere. [↓DP.1] [↓DP.2] [↓DP.3]
Rationale: Because SPHERES is a formation flying testbed, the Spheres must be able to communicate with one another in order to execute maneuvers. In addition, each Sphere needs to be able to download telemetry so that Guest Scientists can analyze the data on the ground.

High-Level Requirements

[FR.1] The Communication Subsystem aboard a Sphere shall send data packets from itself to the Communication Subsystem of another Sphere. [↓DP.2.2] [↓DP.3.2] [↓DP.3.3]
Rationale: The Guest Scientist may want two or more Spheres to communicate directly.

[FR.2] The Communication Subsystem aboard a Sphere shall process incoming data packets from the Communication Subsystem of another Sphere. [↓DP.2.2] [↓DP.3.3]
Rationale: The Guest Scientist may want two or more Spheres to communicate directly.

[FR.3] The Communication Subsystem aboard a Sphere shall send data packets from itself to the SPHERES Laptop. [↓DP.2.2] [↓DP.3.1] [↓DP.3.2]
Rationale: The Sphere must be able to downlink telemetry information to the SPHERES Laptop.

[FR.4] The Communication Subsystem aboard a Sphere shall process incoming data packets, commands and beacon locations from the SPHERES Laptop. [↓DP.2.2] [↓DP.2.5]

Rationale: The Sphere must be able to receive new Guest Scientist Programs and information from the SPHERES Laptop.

[FR.5] The Guest Scientist shall be able to select what telemetry is sent from the Sphere to the SPHERES Laptop. [↓DP.1.2] [↓DP.3.1]
Rationale: Different Guest Scientist may be interested in different data sets.

[FR.6] The Communication Subsystem shall provide a mechanism to share the communication channels. [↓DP.3.4]
Rationale: Each Sphere and the SPHERES Laptop should be able to transmit data packets without interference.

Design and Safety Constraints

Non-Safety Constraints

[SC.1] The Communication Subsystem must operate independently of any operator action. [↓DP.3]
Rationale: The SPHERES system is autonomous and therefore must operate without human interference.

[SC.2] The Communication Subsystem must not operate when it is outside the range of the SPHERES Laptop transmitter.
Rationale: The Russians do not want SPHERES operating outside of the US Node of the ISS.

System Limitations

[L.1] The Communication Subsystem transmits data at 57.6kbps.
Rationale: This is the data transfer rate of the Sphere's radios.

Verification and Validation

Review Procedures

A review board that is independent of the development team verifies levels 1, 2 and 3 of the SPHERES system SpecTRM-GSCs. Multiple iterations of this review process can be performed to help ensure that the analysis reflects the actual operation of the SPHERES system. Suggestions for improvement are added as necessary and evaluated in the next iteration of the review process.

Participants

Kathryn Weiss – SpecTRM-GSC Developer
John Enright – SPHERES Team Member and Primary Reviewer

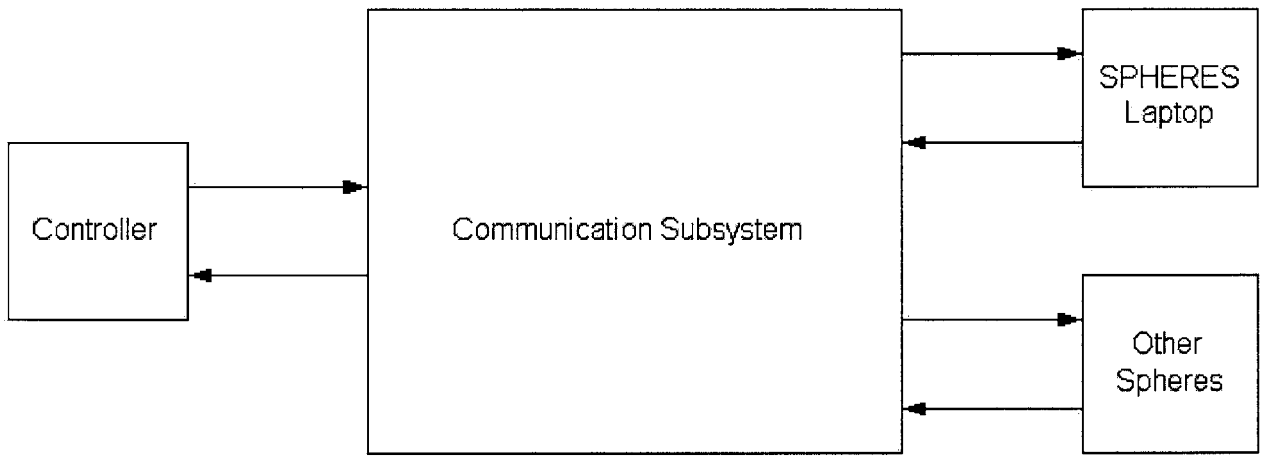
Results

This SpecTRM-GSC is the result of two development and review cycles.

Level 2: System Design Principles

System Interface Design

The external components will interface with the Communication Subsystem in the following manner: [↑EA.1]



The Communication Subsystem interfaces with the Sphere Controller and with the radios that allow the Sphere to communicate with the SPHERES Laptop as well as other Spheres. The radios are not represented in this diagram because they are not a SpecTRM-GSC. They are trivial hardware components and are therefore not modeled.

The Sphere Controller sends the Communication Subsystem telemetry and commands to send to other Spheres. The Communication Subsystem sends commands from other Spheres to the Sphere Controller. [↓C.1] [↓C.2] [↓C.3]

The Communication Subsystem sends telemetry to the SPHERES Laptop. The SPHERES Laptop uploads new Guest Scientist Programs onto the Spheres and also provides the Spheres with beacon locations. [↓C.7] [↓C.8] [↓C.9]

The Communication Subsystem also sends and receives commands from other Spheres. [↓C.4] [↓C.5] [↓C.6]

System Design Principles

[DP.1] Communication Subsystem Overview [↑FG.1]

[DP.1.1] The Communication Subsystem is made up of software and hardware components. The hardware components consist of the I/O Circuitry and the Radios. Figure 1 provides a diagram of the Communication Subsystem.

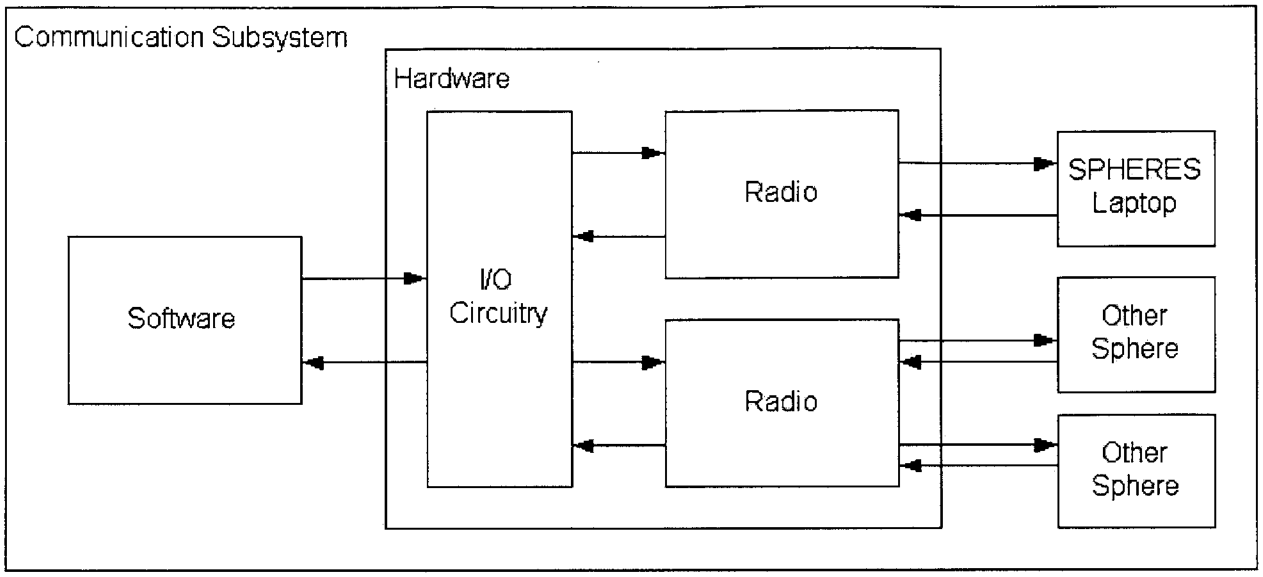


Figure 1. Communication Subsystem Diagram

[DP.1.2] The Guest Scientist will decide whether or not to send telemetry to the SPHERES Laptop. This information is sent through the Sphere Controller to the Communication Subsystem. [↓FR.5] [↓TelemetryFlag]
Rationale: The Guest Scientist may or may not need the data generated by the Guest Scientist Program; therefore he/she needs an option as to whether or not to record telemetry.

[DP.2] Communication Subsystem Hardware [↓FG.1]

[DP.2.1] The I/O Circuitry provides a buffer between the Communication Subsystem and the Radios. It translates the serial data stream from the Communication Subsystem software to the parallel data stream taken by the radios and vice versa.
Rationale: The serial data stream travels much slower than parallel and therefore a buffer is needed between the two streams to accommodate the different flow rates.

[DP.2.2] The Radios transmit and receive data packets to and from the Communications Subsystem, SPHERES Laptop and other Spheres. [↑FR.1] [↑FR.2] [↑FR.3] [↑FR.4]

[DP.2.3] The Radios provide the Communication Subsystem with error detection in the form of checksums.
Rationale: Although the radios do not provide error correction, the Guest Scientist will at least be aware if the collected telemetry has been corrupted.

[DP.2.4] The SPHERES Laptop saves telemetry sent by the Communication Subsystem to a hard disk.
Rationale: The ISS only downlinks data to Earth once every twelve hours.

[2.5] The SPHERES Laptop sends Start Test and Stop Test commands to the Sphere through the Communication Subsystem. [↑FR.4]
Rationale: The Laptop will not start a test until all the information necessary for running the Guest scientist Program has been loaded onto the Spheres.

[DP.3] Communication Subsystem Software [↑FG.1] [↑SC.1]

[DP.3.1] The Communication Subsystem receives standard state and engineering telemetry as well as user defined experimental data from the Sphere Controller. If the Sphere Controller also sends a command that the telemetry is to be recorded, the Communication Subsystem sends the telemetry to the SPHERES Laptop. [↑FR.3] [↑FR.5]
Rationale: The Guest Scientist may wish to gather data outside the standard state and engineering telemetry provided by the Sphere Controller.

[DP.3.2] The Communication Subsystem generates telemetry packets to be sent to the Radios. [↑FR.1] [↑FR.3]

[DP.3.3] The Communication Subsystem sends information and/or commands from the Sphere Controller to other Spheres. [↑FR.1] [↑FR.2]
Rationale: Guest Scientist Programs that require multiple Spheres may need the Spheres to send commands to one another and to share data.

[DP.3.4] The Communication Subsystem provides Time Division Multiple Access (TDMA) for the Sphere. [↑FR.6]
Rationale: TDMA allows the SPHERES system to share the two radio channels among the multiple stations, giving each station the opportunity to transmit.

Level 3: Blackbox Behavior

Communication

[C.1] The Sphere Controller sends the Communication Subsystem a Boolean telemetry flag, indicating whether or not to send telemetry. [↑System Interface Design] [→TelemetryFlagInput]

[C.2] The Sphere Controller sends the Communication Subsystem the state vector (Angular Rate (x, y, z), Position (x, y, z), Linear Velocity (x, y, z) and Quaternion (q1, q2, q3, q4)) as well as any other telemetry specified by the Guest Scientist. [↑System Interface Design] [→VelocityXInput] [→VelocityYInput] [→VelocityZInput] [→AngularRateXInput] [→AngularRateYInput] [→AngularRateZInput]

[C.3] The Sphere Controller sends the Communication Subsystem commands for the Other Spheres. [↑System Interface Design]

[C.4] The Communication Subsystem sends commands to Other Spheres. [↑System Interface Design] [→RateMatcherData]

[C.5] The Communication Subsystem receives commands from Other Spheres. [↑System Interface Design]

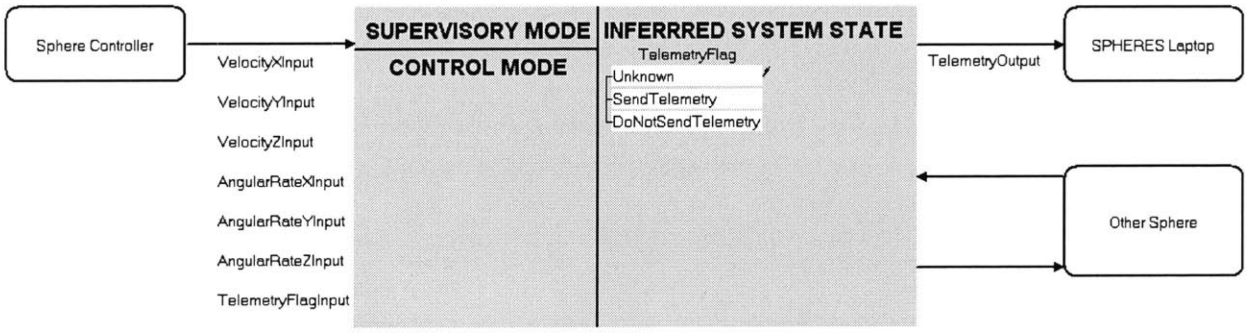
[C.6] The Communication Subsystem sends telemetry to the SPHERES Laptop. [↑System Interface Design] [→TelemetryOutput]

[C.7] The SPHERES Laptop sends Guest Scientist Programs to the Communication Subsystem. [↑System Interface Design]

[C.8] The SPHERES Laptop sends Start Test and Stop Test commands to the Communication Subsystem. [↑System Interface Design]

[C.9] The SPHERES Laptop sends Beacon Locations to the Communication Subsystem. [↑System Interface Design]

System Blackbox Behavior – Communication Subsystem



Appendix D

Sphere Controller

Level 1: System-Level Goals, Requirements, and Constraints

Introduction

The Sphere Controller aboard each Sphere in the SPHERES system (Synchronize Position Hold Engage Reorient Experimental Satellites) provides the overall framework for the Sphere. The Sphere Controller coordinates the actions of the onboard components as well as determining the operating mode of the Sphere.

Historical Information

This intent specification is a SpecTRM-GSC (Generic Spacecraft Component). It should only be used for the SPHERES project and encompasses only information relating to the Sphere Controller.

Environment Description

There are six subsystems (Propulsion Subsystem, PADS, Communication Subsystem, Guest Scientist Program, Structure and Electrical Subsystem) in the Sphere Controller's environment. The Sphere Controller communicates with only four of these subsystems:

Propulsion Subsystem – The Propulsion Subsystem provides both position and attitude management for the Sphere. It is a subsystem-level intent specification in the SpecTRM-GSC decomposition of SPHERES.

PADS – PADS provides state estimation for the Sphere. It is a subsystem-level intent specification in the SpecTRM-GSC decomposition of SPHERES.

Communication Subsystem – The Communication Subsystem provides wireless data transfer for the SPHERES system. It is a subsystem-level intent specification in the SpecTRM-GSC decomposition of SPHERES.

Guest Scientist Program – The Guest Scientist Program allows scientists outside of MIT and NASA to test high-risk metrology, control and autonomy algorithms in the environment of micro-gravity. Each Guest Scientist Program is a different subsystem-level intent specification in the SpecTRM-GSC decomposition of SPHERES.

Environment Assumptions

[EA.1] The Sphere Controller is operating within a Sphere. [↓System Interface Design]
Rationale: This intent specification describes the Controller that will operate within a Sphere. It is not meant for use outside the SPHERES project.

[EA.2] The Sphere Controller is operating within a Sphere that contains a Propulsion Subsystem, PADS, Communication Subsystem and Guest Scientist Program designed for the SPHERES project. [↓System Interface Design]
Rationale: The Sphere Controller is not a generic controller and is meant to be used with subsystems designed for the SPHERES project.

Environment Constraints

[EC.1] The Sphere must only operate within Node 1 of the International Space Station.
Rationale: The Russians do not want SPHERES operating outside of the US Node of the ISS.

System Goals

[FG.1] The Sphere Controller shall coordinate the actions of the Sphere's onboard components. [→FR.1] [→FR.2] [→FR.3] [↓DP.1] [↓DP.2] [↓DP.3] [↓DP.4]
Rationale: A controller is needed to coordinate the actions of the four active subsystems onboard the Sphere.

High-Level Requirements

[FR.1] The Sphere Controller shall execute the Guest Scientist Program. [→FG.1] [↓DP.1] [↓DP.4.5]
Rationale: The Guest Scientist Program contains the control laws and/or state estimation algorithms that allow the Spheres to perform maneuvers. Running an experiment involves the Sphere Controller executing one of these programs.

[FR.2] The Sphere Controller shall manage the interactions between subsystems. [→FG.1] [↓DP.1] [↓DP.2] [↓DP.3]
Rationale: All interactions between the subsystems are channeled through the Sphere Controller.

[FR.3] The Sphere Controller shall provide the overall operating mode of the Sphere. [→FG.1] [↓DP.4]
Rationale: Because the Sphere Controller executes the Guest Scientist Program, it is aware of the status of the experiment and therefore sets the operating mode.

Design and Safety Constraints

Non-Safety Constraints

[SC.1] The Sphere Controller must operate independently of any operator action. [↓DP.1] [↓DP.2] [↓DP.3]
Rationale: The SPHERES system is autonomous and therefore must operate without human interference.

Operator Requirements

[OR.1] The Operator shall enable the Sphere Controller after the Guest Scientist Program has been loaded and the test area prepared. [↓OT.1] [↓OT.2] [↓OT.3]
Rationale: The test area must be clear of astronauts and other objects foreign to the SPHERES system before the Guest Scientist Program is executed. The Guest Scientist Program cannot start unless the Sphere Controller has been enabled.

[OR.2] The Operator shall start the SPHERES experiment after the Sphere Controller has been enabled and stop the experiment when the Guest Scientist Program is finished executing. [↓OT.4]

Verification and Validation

Review Procedures

A review board that is independent of the development team verifies levels 1, 2 and 3 of the SPHERES system SpecTRM-GSCs. Multiple iterations of this review process can be performed to help ensure that the analysis reflects the actual operation of the SPHERES system. Suggestions for improvement are added as necessary and evaluated in the next iteration of the review process.

Participants

Kathryn Weiss – SpecTRM-GSC Developer
John Enright – SPHERES Team Member and Primary Reviewer

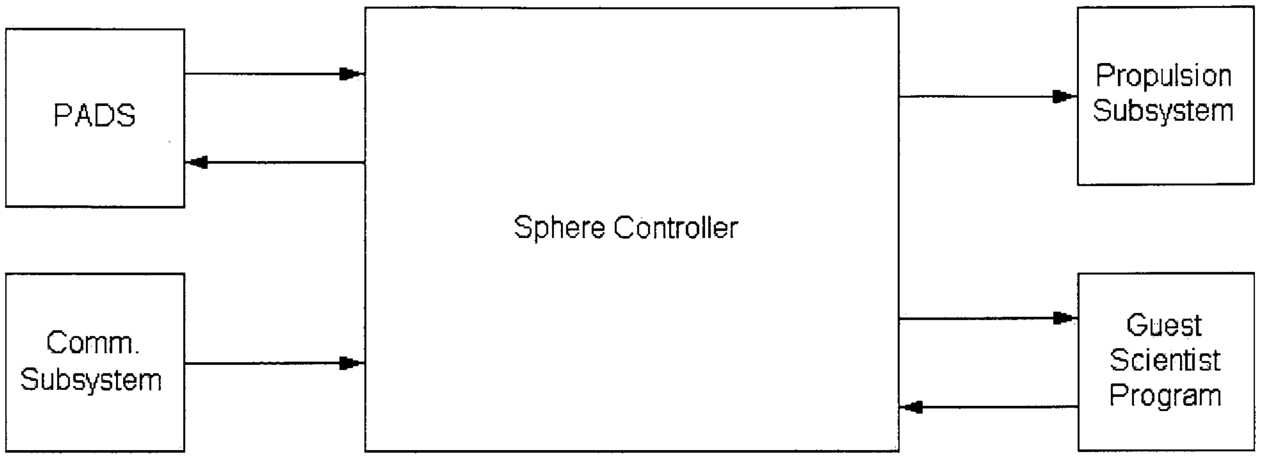
Results

This SpecTRM-GSC is the result of two development and review cycles.

Level 2: System Design Principles

System Interface Design

The four external subsystems will interface with the Sphere Controller in the following manner: [↑EA.1] [↑EA.2]



The Sphere Controller communicates with the four active subsystems: PADS, Communication Subsystem, Propulsion Subsystem and Guest Scientist Program.

The Sphere Controller tells PADS which information is needed from the avionics hardware. The Sphere Controller can either receive linear acceleration and/or angular velocity from the inertial measurement system and/or ranges to each beacon from the global measurement system. PADS sends the requested information to the Sphere Controller. [↓C.2]

The Sphere Controller sends the Communication Subsystem telemetry and commands to send to other Spheres. The Communication Subsystem sends commands from other Spheres to the Sphere Controller. [↓C.3]

The Sphere Controller sends the Propulsion Subsystem either timed on/off commands for each of the twelve thrusters or force and torque vectors. The Propulsion Subsystem does not send any information to the Sphere Controller. [↓C.1]

The Sphere Controller sends and receives different information from the Guest Scientist Program depending on the needs of the Guest Scientist. The Sphere Controller will always receive the control mode for the Propulsion Subsystem and PADS from the Guest Scientist Program. Other information is dependent on the specific program being executed. A detailed description of the information exchanged is found at Level 3 of this model. [↓C.4]

Operator Task Design Principles

[OT.1] The Operator loads the Guest Scientist Program onto the Sphere through the SPHERES Laptop. [↑OR.1] [↓OP.4]

[OT.2] The Operator prepares the test area for running a SPHERES experiment. [↑OR.1] [↓OP.1] [↓OP.2] [↓OP.3] [↓OP.5]

[OT.3] The Operator enables the Sphere Controller by pushing an enable button on the Sphere. [↑OR.1] [↓OP.6]

[OT.4] The Operator starts and stops the SPHERES experiment through the SPHERES Laptop. [↑OR.2] [↓OP.7] [↓OP.8]

System Design Principles

[DP.1] The Sphere Controller provides the Guest Scientist Program with position and attitude information from PADS. [↑FG.1] [↑FR.1] [↑FR.2] [↑SC.1]
Rationale: The Guest Scientist Program requires position and attitude information to determine desired forces and torques to accomplish a maneuver.

[DP.2] The Sphere Controller provides the Propulsion Subsystem with either direct "On Time" and "Off Time" commands or force and torque vectors calculated by the Guest Scientist Program. [↑FG.1] [↑FR.2] [↑SC.1] [↓PropulsionSubsystemState]

[DP.3] The Sphere Controller generates automatic telemetry from the position and attitude information provided by PADS and sends it to the Communication Subsystem. The Sphere Controller also sends information specified by the Guest Scientist Program to the Communication Subsystem. [↑FG.1] [↑FR.2] [↑SC.1] [↓PADSState]

[DP.4] Sphere Controller Operating Modes [↑FG.1] [↑FR.3] [↓SphereControlMode]

[DP.4.1] The Sphere Controller enters "Boot" when the Sphere is Reset or Powered On.

[DP.4.2] The Sphere Controller enters mode "Load Program" if there is a new Guest Scientist Program to load onto the Sphere.

[DP.4.3] The Sphere Controller enters "Idle" if the Sphere successfully loaded the Guest Scientist Program or if there was no new Guest Scientist Program to load onto the Sphere from mode "Boot"

[DP.4.4] The Sphere Controller enters mode "Position Hold" from "Idle" when the Operator enables the push button on the Sphere.

[DP.4.5] When the Sphere Controller receives a "Start" command from the SPHERES Laptop through the Communication Subsystem, the Sphere enters mode "User Control" which signals that the Guest Scientist Program is running and controlling the Sphere. [↑FR.1]

[DP.4.6] If the Sphere Controller receives a "Stop" command from the SPHERES Laptop through the Communication Subsystem, the Sphere goes back to "Idle" mode.

Level 3: Blackbox Behavior

Communication

[C.1] Propulsion Subsystem [↑System Interface Design]

[C.1.1] The Sphere Controller sends the Propulsion Subsystem the operating mode specified by the Guest Scientist Program. [→PropulsionSubsystemModeOutput]

[C.1.2] The Sphere Controller sends the Propulsion Subsystem a Forces and Torques Vector containing the desired forces (x, y, z) and torques (x, y, z) required by the Guest Scientist Program. [→ForceTorqueVectorOutput]

[C.2] PADS [↑System Interface Design]

[C.2.1] The Sphere Controller sends PADS the operating mode specified by the Guest Scientist Program. [→PADSModeOutput]

[C.2.2] The Sphere Controller receives from PADS either:

[C.2.2.1] Ranges from each of the 24 ultrasound receivers to each of the 5 beacons in meters.

[C.2.2.2] Angular Velocity (x, y, z) in radians per second and Linear Acceleration (x, y, z) in meters per second squared. [→AngularRateXInput] [→AngularRateYInput] [→AngularRateZInput]

[C.2.2.3] Both ranges from each of the 24 ultrasound receivers to each of the 5 beacons in meters and Angular Velocity (x, y, z) in radians per second and Linear Acceleration (x, y, z) in meters per second squared.

[C.3] Communication Subsystem [↑System Interface Design]

[C.3.1] The Sphere Controller sends the Communication Subsystem a Boolean telemetry flag, indicating whether or not to send telemetry. [→TelemetryFlagOutput]

[C.3.2] The Sphere Controller sends the Communication Subsystem the state vector (Angular Rate (x, y, z), Position (x, y, z), Linear Velocity (x, y, z) and Quaternion (q1, q2, q3, q4)) as well as any other telemetry specified by the Guest Scientist. [→AngularRateOutput]

[C.3.3] The Sphere Controller sends the Communication Subsystem commands and/or information for the Other Spheres.

[C.4] Guest Scientist Program [↑System Interface Design]

[C.4.1] The Guest Scientist Program sends the desired operating mode for the Propulsion Subsystem to the Sphere Controller. [→PropulsionSubsystemModeInput]

[C.4.2] The Guest Scientist Program sends the desired operating mode for PADS to the Sphere Controller. [→PADSModeInput]

[C.4.3] Rate Damper

[C.4.3.1] The Sphere Controller sends the Guest Scientist Program Angular Rates (x, y, z). [→AngularRateXOutput] [→AngularRateYOutput] [→AngularRateZOutput]

[C.4.3.2] The Rate Damper sends the Sphere Controller desired forces (x, y, z) and torques (x, y, z) needed from the Propulsion Subsystem to cancel the angular rate. [→ForceXInput] [→ForceYInput] [→ForceZInput] [→TorqueXInput] [→TorqueYInput] [→TorqueZInput]

[C.4.4] Rate Matcher

[C.4.4.1] The Sphere Controller sends the Guest Scientist Program Angular Rates (x, y, z). [→AngularRateXOutput] [→AngularRateYOutput] [→AngularRateZOutput]

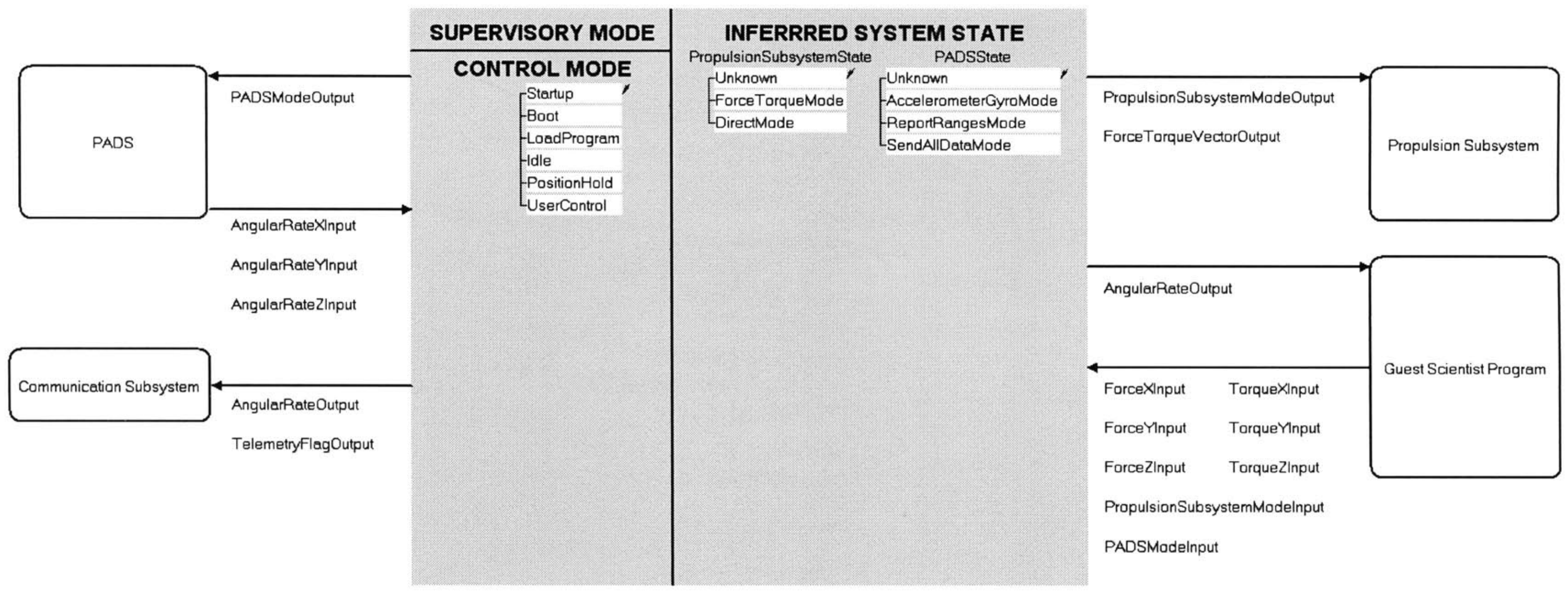
[C.4.4.2] The Sphere Controller sends the Guest Scientist Program the Angular Rates (x, y, z) of the Leader Sphere. [→LeaderAngularRateXOutput] [→LeaderAngularRateYOutput] [→LeaderAngularRateZOutput]

[C.4.4.3] The Rate Damper sends the Sphere Controller desired forces (x, y, z) and torques (x, y, z) needed from the Propulsion Subsystem to cancel the angular rate. [→ForceXInput] [→ForceYInput] [→ForceZInput] [→TorqueXInput] [→TorqueYInput] [→TorqueZInput]

Operational Procedures

[OP.1] Unstow Sphere(s). [↑OT.2]
[OP.2] Check the amount of propellant in each Sphere tank. If the amount is below 20%, follow Operational Procedures for the Propulsion Subsystem. [↑OT.2]
[OP.3] Follow the Operational Procedures [OP.1] and [OP.2] for PADS. [↑OT.2]
[OP.4] Load a Guest Scientist Program on the Sphere Controller through the SPHERES Laptop. [↑OT.1]
[OP.5] Place the Sphere(s) in the middle of the test area. [↑OT.2]
[OP.6] Press the enable button on the Sphere(s). [↑OT.3]
[OP.7] Send "Start" command through the SPHERES Laptop. [↑OT.4]
[OP.8] After the test is completed send "Stop" command through the SPHERES Laptop. [↑OT.2]
[OP.9] Stow Sphere(s).
[OP.10] Follow the Operational Procedures [OP.3] for PADS.

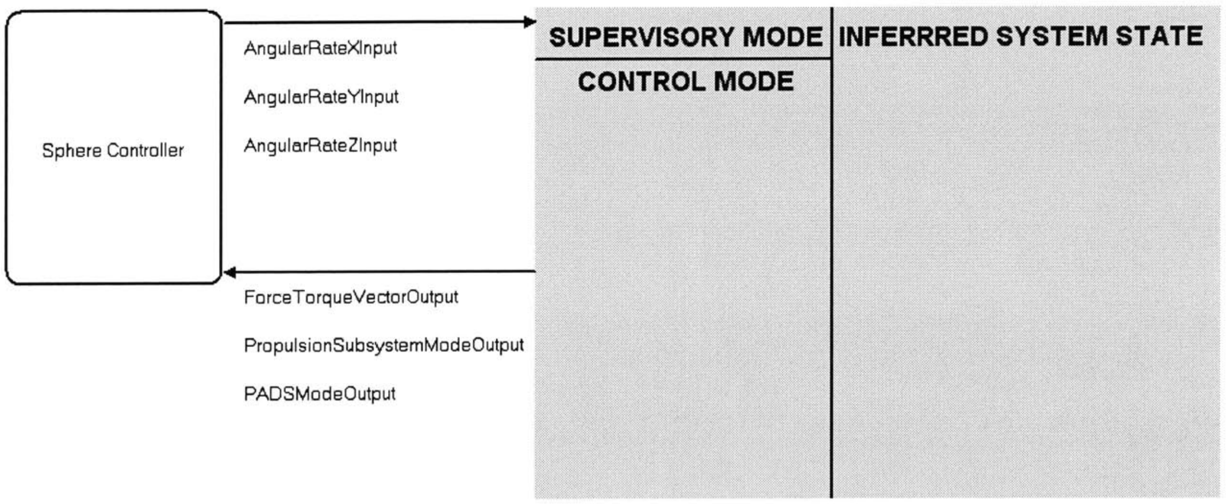
System Blackbox Behavior – Sphere Controller



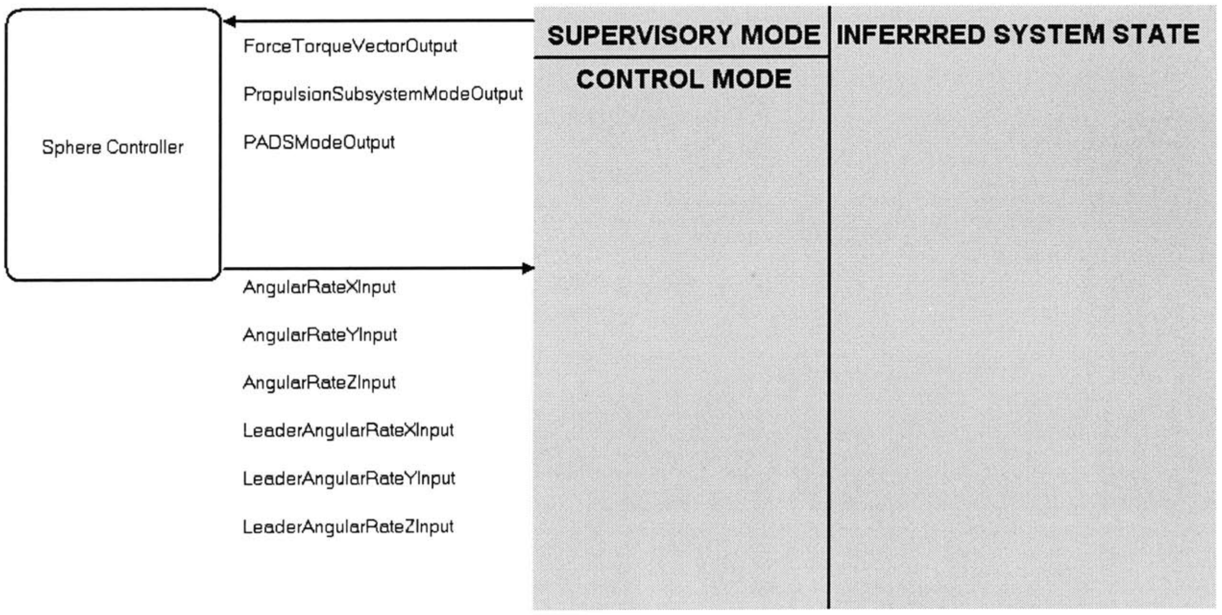
Appendix E

System Blackbox Behavior – Guest Scientist Programs

Rate Damper



Rate Matcher



Appendix F

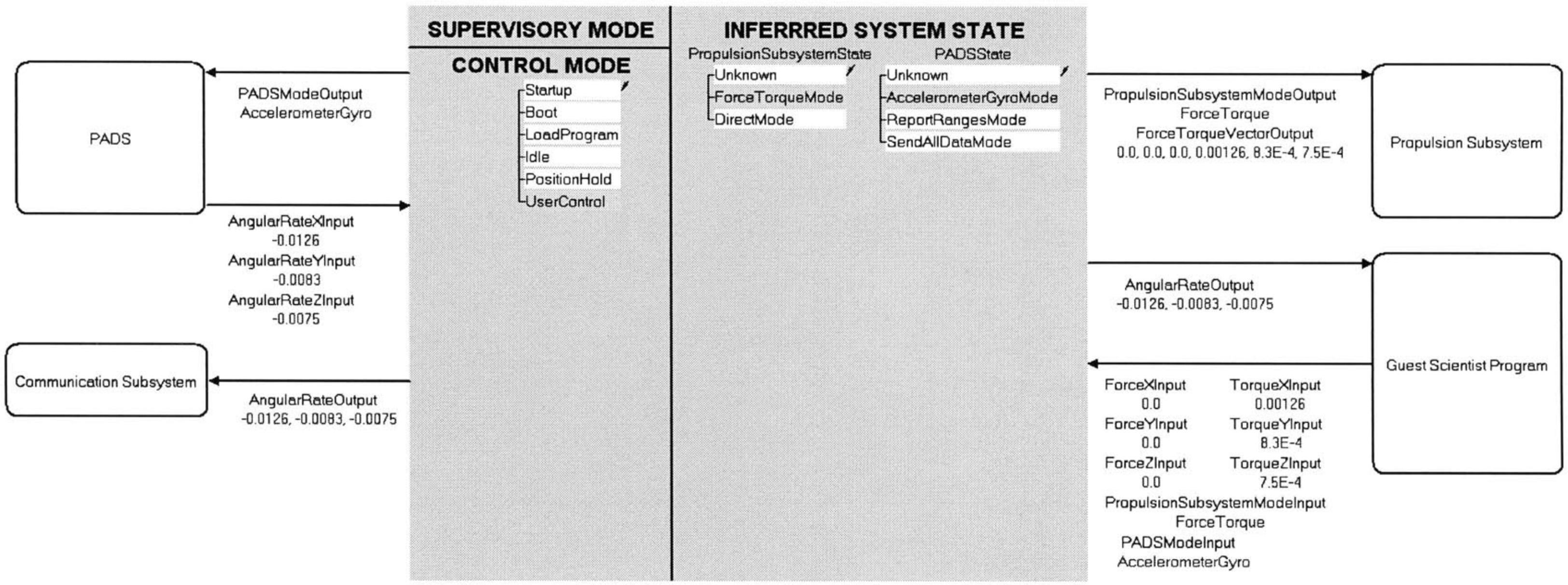
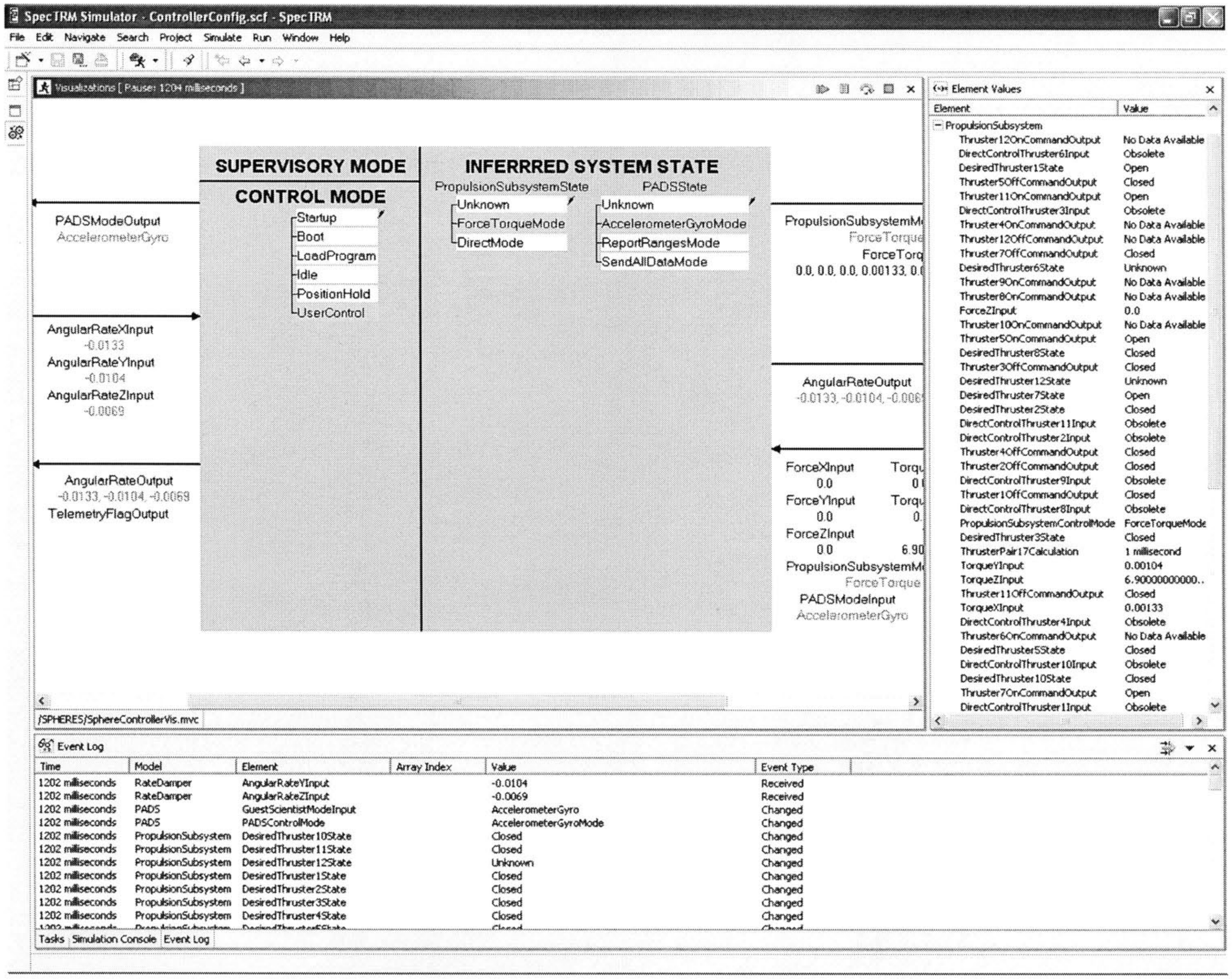


Figure 1. Sphere Controller Blackbox Diagram During Rate Damper Simulation



Element Values

Element	Value
− PropulsionSubsystem	
Thruster12OnCommandOutput	No Data Available
DirectControlThruster6Input	Obsolete
DesiredThruster1State	Open
Thruster5OffCommandOutput	Closed
Thruster11OnCommandOutput	Open
DirectControlThruster3Input	Obsolete
Thruster4OnCommandOutput	No Data Available
Thruster12OffCommandOutput	No Data Available
Thruster7OffCommandOutput	Closed
DesiredThruster6State	Unknown
Thruster9OnCommandOutput	No Data Available
Thruster8OnCommandOutput	No Data Available
ForceZInput	0.0
Thruster10OnCommandOutput	No Data Available
Thruster5OnCommandOutput	Open
DesiredThruster8State	Closed
Thruster3OffCommandOutput	Closed
DesiredThruster12State	Unknown
DesiredThruster7State	Open
DesiredThruster2State	Closed
DirectControlThruster11Input	Obsolete
DirectControlThruster2Input	Obsolete
Thruster4OffCommandOutput	Closed
Thruster2OffCommandOutput	Closed
DirectControlThruster9Input	Obsolete
Thruster1OffCommandOutput	Closed
DirectControlThruster8Input	Obsolete
PropulsionSubsystemControlMode	ForceTorqueMode
DesiredThruster3State	Closed
ThrusterPair17Calculation	1 millisecond
TorqueYInput	0.00104
TorqueZInput	6.9000000000..
Thruster11OffCommandOutput	Closed
TorqueXInput	0.00133
DirectControlThruster4Input	Obsolete
Thruster6OnCommandOutput	No Data Available
DesiredThruster5State	Closed
DirectControlThruster10Input	Obsolete
DesiredThruster10State	Closed
Thruster7OnCommandOutput	Open
DirectControlThruster1Input	Obsolete

Event Log

Time	Model	Element	Array Index	Value	Event Type
1202 milliseconds	RateDamper	AngularRateYInput		-0.0104	Received
1202 milliseconds	RateDamper	AngularRateZInput		-0.0069	Received
1202 milliseconds	PADS	GuestScientistModeInput		AccelerometerGyro	Changed
1202 milliseconds	PADS	PADSControlMode		AccelerometerGyroMode	Changed
1202 milliseconds	PropulsionSubsystem	DesiredThruster10State		Closed	Changed
1202 milliseconds	PropulsionSubsystem	DesiredThruster11State		Closed	Changed
1202 milliseconds	PropulsionSubsystem	DesiredThruster12State		Unknown	Changed
1202 milliseconds	PropulsionSubsystem	DesiredThruster1State		Closed	Changed
1202 milliseconds	PropulsionSubsystem	DesiredThruster2State		Closed	Changed
1202 milliseconds	PropulsionSubsystem	DesiredThruster3State		Closed	Changed
1202 milliseconds	PropulsionSubsystem	DesiredThruster4State		Closed	Changed
1202 milliseconds	PropulsionSubsystem	DesiredThruster5State		Closed	Changed

Figure 2. Screen Capture of the Simulation Environment for the Rate Damper Example

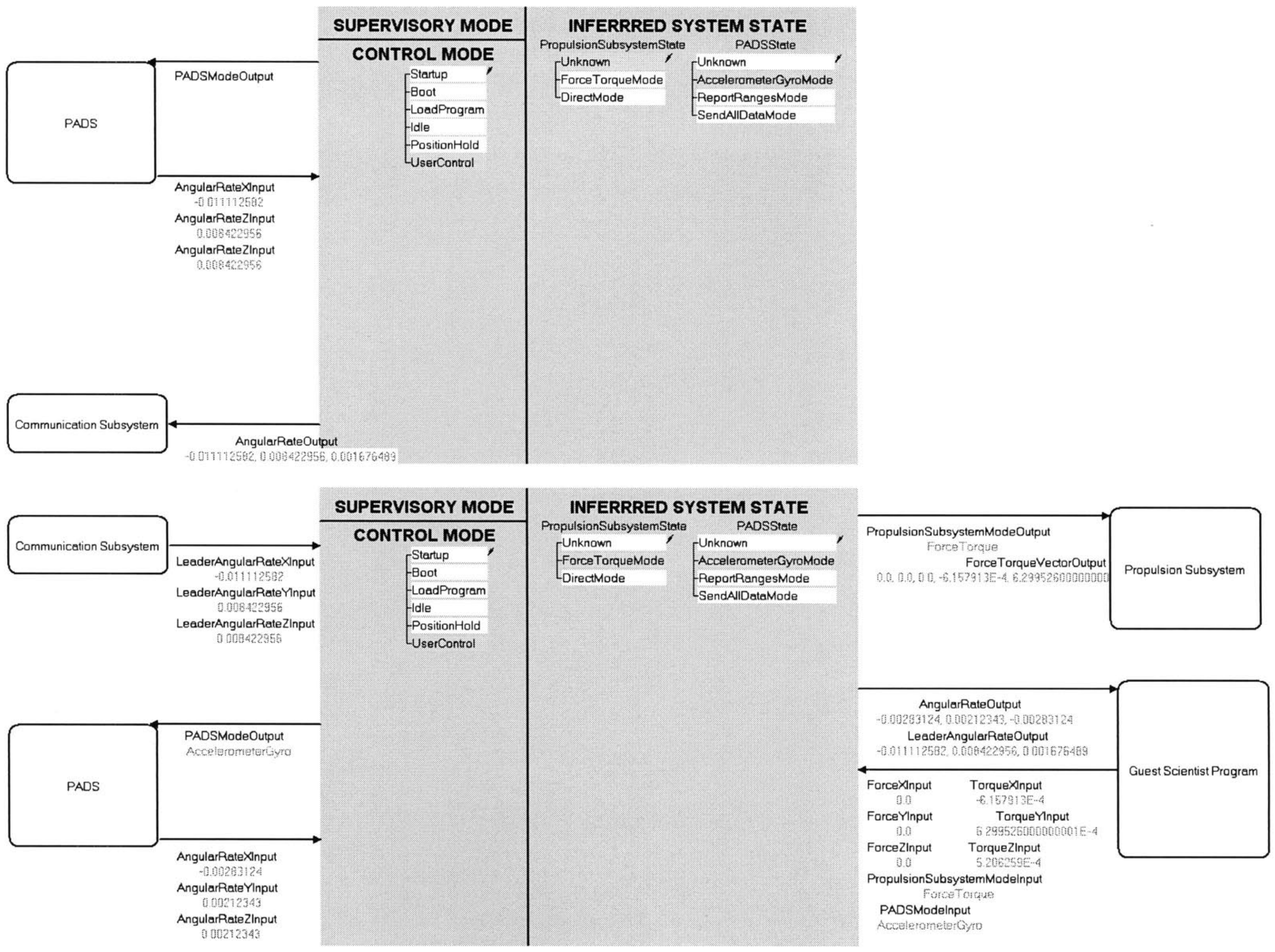
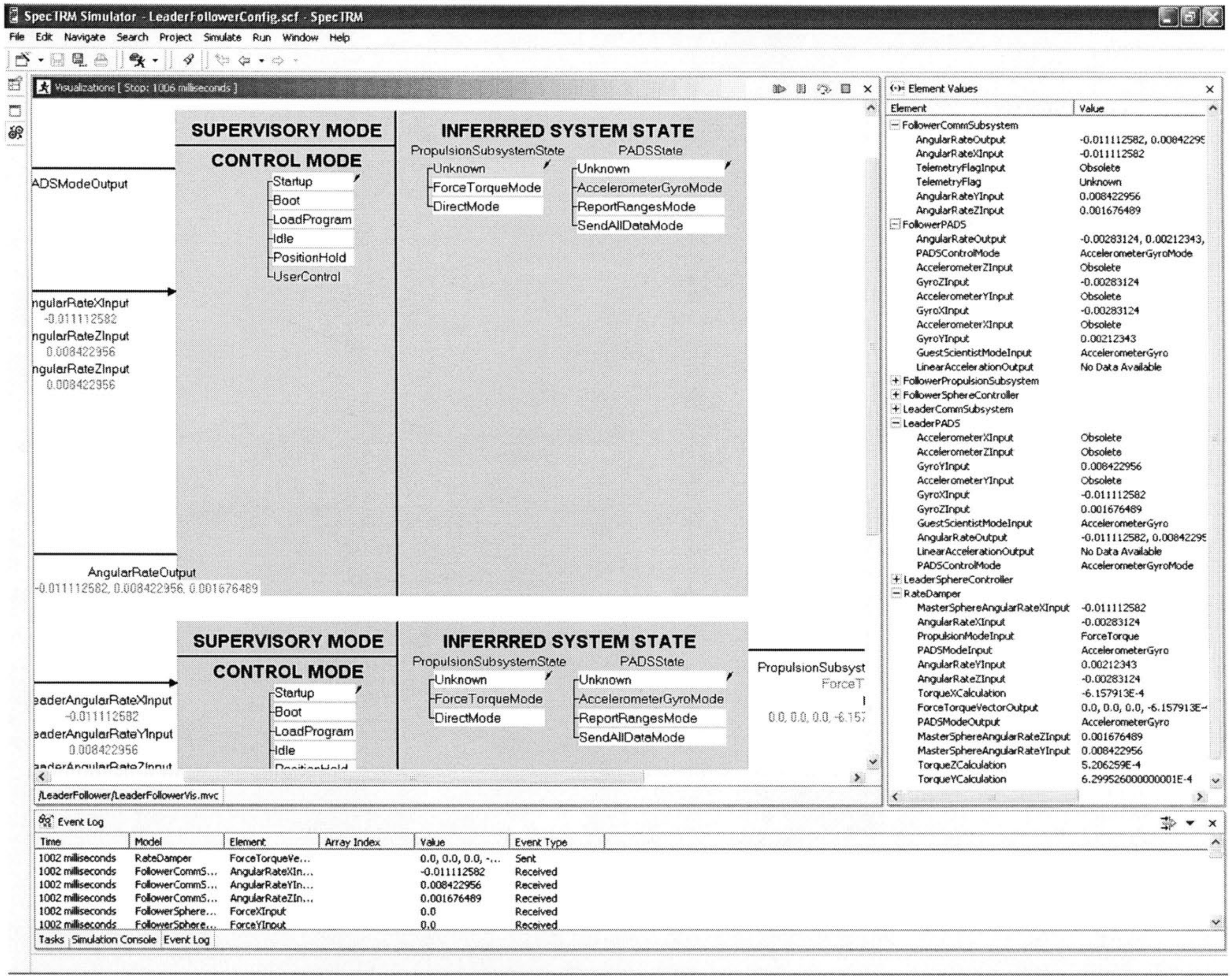


Figure 3. Leader and Follower Sphere Controller Blackbox Diagrams During Rate Matcher Simulation



Element	Value
− FollowerCommSubsystem	
AngularRateOutput	-0.011112582, 0.00842295
AngularRateXInput	-0.011112582
TelemetryFlagInput	Obsolete
TelemetryFlag	Unknown
AngularRateYInput	0.008422956
AngularRateZInput	0.001676489
− FollowerPADS	
AngularRateOutput	-0.00283124, 0.00212343,
PADSControlMode	AccelerometerGyroMode
AccelerometerZInput	Obsolete
GyroZInput	-0.00283124
AccelerometerYInput	Obsolete
GyroXInput	-0.00283124
AccelerometerXInput	Obsolete
GyroYInput	0.00212343
GuestScientistModeInput	AccelerometerGyro
LinearAccelerationOutput	No Data Available
+ FollowerPropulsionSubsystem	
+ FollowerSphereController	
+ LeaderCommSubsystem	
− LeaderPADS	
AccelerometerXInput	Obsolete
AccelerometerZInput	Obsolete
GyroYInput	0.008422956
AccelerometerYInput	Obsolete
GyroXInput	-0.011112582
GyroZInput	0.001676489
GuestScientistModeInput	AccelerometerGyro
AngularRateOutput	-0.011112582, 0.00842295
LinearAccelerationOutput	No Data Available
PADSControlMode	AccelerometerGyroMode
+ LeaderSphereController	
− RateDamper	
MasterSphereAngularRateXInput	-0.011112582
AngularRateXInput	-0.00283124
PropulsionModeInput	ForceTorque
PADSModeInput	AccelerometerGyro
AngularRateYInput	0.00212343
AngularRateZInput	-0.00283124
TorqueXCalculation	-6.157913E-4
ForceTorqueVectorOutput	0.0, 0.0, 0.0, -6.157913E-
PADSModeOutput	AccelerometerGyro
MasterSphereAngularRateZInput	0.001676489
MasterSphereAngularRateYInput	0.008422956
TorqueZCalculation	5.206259E-4
TorqueYCalculation	6.299526000000001E-4

Time	Model	Element	Array Index	Value	Event Type
1002 milliseconds	RateDamper	ForceTorqueVe...		0.0, 0.0, 0.0, -...	Sent
1002 milliseconds	FollowerCommS...	AngularRateXIn...		-0.011112582	Received
1002 milliseconds	FollowerCommS...	AngularRateYIn...		0.008422956	Received
1002 milliseconds	FollowerCommS...	AngularRateZIn...		0.001676489	Received
1002 milliseconds	FollowerSphere...	ForceXInput		0.0	Received
1002 milliseconds	FollowerSphere...	ForceYInput		0.0	Received

Figure 4. Screen Capture of the Simulation Environment for the Rate Matcher Example